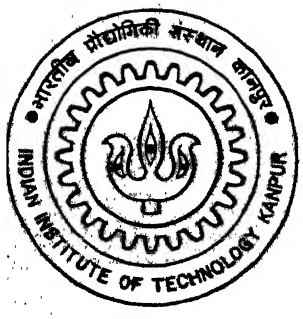
CODE GENERATOR FOR CONVEX C2

by

NARASIMHAN G.



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

October, 1994

CODE GENERATOR FOR CONVEX C220

A thesis submitted

in partial fulfilment of the requirements

for the degree of

Master of Technology

by

Narasimhan G

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

October, 1994

CSE-1994-M-NAR-COD

CERTIFICATE

It is certified that the work contained in the thesis titled *CODE GENERATOR FOR CONVEX C220*, by Narasimhan G., has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

October, 1994

Sanjeev Kumar Aggarwal
Associate Professor,
Department of Computer Science
and Engineering,
IIT, Kanpur.

ABSTRACT

In this thesis we deal with the design and implementation of a code generator for CONVEX C220, an architecture which supports vector processing. The work is part of the ongoing FORTRAN 90 vectorizing compiler project. The code generator generates CONVEX C220 assembly code from a low-level machine dependent intermediate representation. This intermediate representation was designed. Specifically, the code generator does instruction selection and register allocation. The instruction selector is based on a retargetable parser driven pattern matching method and is derived automatically from a description of the target machine using a parser generator. The target machine description is in the form of a context free grammar. The target machine description grammar was designed. The register allocator implements an on-the-fly register allocation strategy augmented with a facility for specifying use counts for values. By virtue of the method used, it is easy to ensure the correctness of the code generator and to tune and maintain the code generator.

ACKNOWLEDGEMENTS

I am grateful to my thesis supervisor Dr. Sanjeev Kumar Aggarwal for his invaluable help and guidance during the thesis work and in the preparation of this report. I thank him for his patience in helping me overcome the problems I faced, both academic and otherwise.

My special thanks to Vaidya who went out of his ways to help me on many an occasion in spite of his own problems. I recall the wonderful and lively company of my batch-mates GCP, Murali and Sai. GCP and Murali were always there to help me. I remember the "music sessions" I used to have with GCP (who removed my misconceptions about Hindustani music), Hari and Sai (the "Swara" parser). Thanks to Sajith, Apu, Kishore and Jayakumar for their friendship and help.

I wish to thank my parents and brothers for their concern and support and for having put up with long periods of silence from me. Thanks to my uncle who has been a source of inspiration and encouragement in all my academic pursuits.

Contents

List of Figures

Chapter 1

Introduction

There is an increasing demand for high-performance computers in many scientific and engineering applications. Recent advances in computer architecture has produced vector processors which provide a means of satisfying this demand. On the software front the effort is to reflect the power of vector processors at the programming language level. One such effort is the FORTRAN 90 language which provides vector constructs and operations. Since much of the scientific and engineering software written so far is in conventional (i.e., scalar) FORTRAN, there is a need as well as motivation to write a vectorizing compiler which would take scalar FORTRAN code and generate equivalent code for a vector computer. The ongoing FORTRAN 90 compiler project at Indian Institute of Technology, Kanpur, is such an effort aimed at developing a vectorizing compiler which would convert sequential programs into vectorized FORTRAN 90 programs and then generate vector code for the CONVEX C220 vector computer.

1.1 Organization of the compiler

This section gives a brief description of the logical structure of the FORTRAN 90 compiler being developed. The compiler has been organized as a set of logical phases each doing a subtask of the complex process of compilation. The phases operate sequentially on a program to be compiled. Each phase transforms the input program in various ways to finally produce the target code. The phases involved and their order of application are shown in figure 1.1. The purpose of each phase is outlined below.

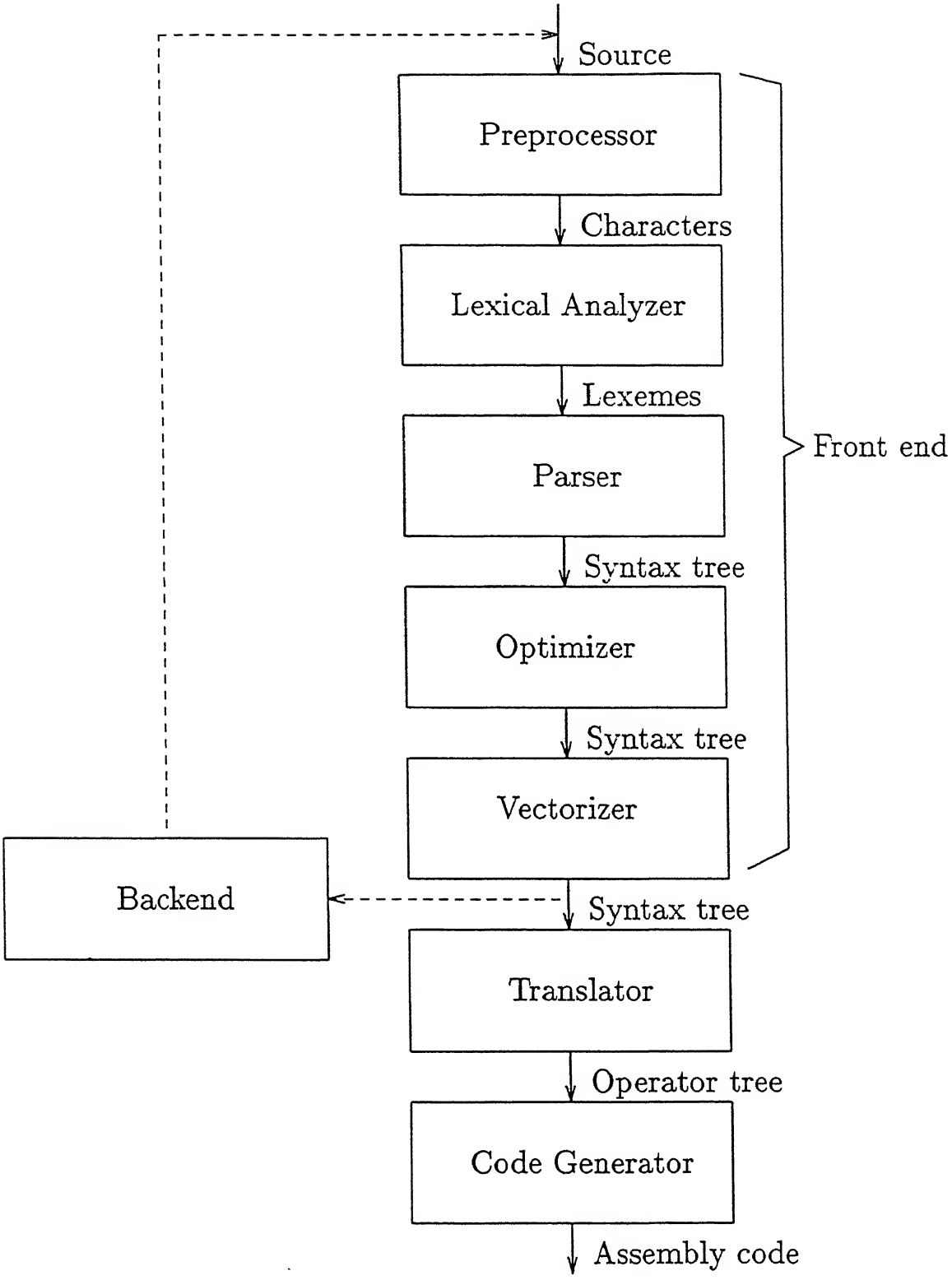


Figure 1.1: The logical structure of the FORTRAN 90 compiler.

Preprocessor : The source program is input to the preprocessor. The preprocessor reads the source file(s) and handles certain compiler directives such as file inclusion and macro expansion. The output of this phase is the fully expanded FORTRAN program.

Lexical analyzer: The output of the preprocessor is the input to the lexical analyzer. It splits the input program text into lexical units called lexemes or tokens.

Parser: The parser's job is to group the lexemes into syntactic units of FORTRAN 90. If the program is free of syntactic errors, the parser produces a tree representation of the program called abstract syntax tree. In this representation each internal node of the tree represents an operator in FORTRAN 90 or a control construct such as the if-then statement. The subnodes of a node represent the operands of the operation. For example, for an assignment statement the node represents the assignment operator and there are two subtrees which correspond to the left and right operands of the assignment. For a control statement like if-then, the node represents the if-then statement and the subtrees represent the condition being checked and the statement body to be executed if the condition is true. The subtrees in turn are structured in a similar manner. A symbol table is also constructed which contains information about the names (such as variables, subroutine names etc.,) used in the program.

Optimizer: This is an optional phase that does possible code improving transformations on the syntax tree produced by the previous phase. This is done by gathering information about the control flow and data flow within the program by doing an analysis of the program. This information is used to restructure the input syntax tree. The output of this phase is also a syntax tree. This syntax tree is semantically equivalent to the input syntax tree, but it may run faster and/or occupy less memory than the input syntax tree. It is possible to include this phase selectively using flags specified when the compiler is invoked. Typically, this phase is invoked only after the programmer has made all modifications to his/her program so that it does not have any syntactic errors and the logic is correct. This is done to save compilation time during the development of the program.

Many transformations are done by this phase. Some of the typical transformations done include the following. *Common subexpressions* which are used more than once are stored and then reused to avoid recomputation (*common subexpression elimination*),

constant expressions are precomputed (*constant folding*), loop invariant computations are moved out of loops (*code motion*), variables which hold constant values are replaced by the constant itself (*constant propagation*).

Vectorizer: This phase takes the syntax tree output of the previous phase and attempts to replace sequences of scalar operations by vector operations. A typical example is to convert a DO-loop into a vector operation. The output of this phase is also a syntax tree.

Backend: From the syntax tree output of the vectorizer this phase can generate an equivalent FORTRAN 90 program. This process is called unparsing. The syntax tree nodes maintain enough information to recreate the source program with modifications equivalent to those done by the optimizer and the vectorizer. This helps the programmer in seeing the transformations performed on the original program. This phase is typically used during the program development process. The programmer may make his own changes in the program and feed it back to the compiler. Usually these manual changes are done with a view to make possible more optimizations and/or vectorizations, which the compiler may otherwise not do as it may not have the necessary information to convince itself that these transformations do preserve the logic of the program.

Translator: The translator maps the syntax tree output of its previous phase to a low level internal representation . This representation consists of operators and operand types that are recognized by the target machine instruction set. All source level data types are translated to equivalent machine-level data types. Source level operations are converted into equivalent machine-level operations or sequences of operations. Complex source level operations like array and record references are broken down to lower level operations which explicate the address computation involved in these references. Control constructs like IF-THEN and DO-LOOP are broken down into a sequence of tests and jumps. Implicit aspects of the source language are made explicit and if evaluation order is unspecified, conventions are set up. Memory allocation is done for the various data objects used in the program. A variable is allocated storage in the static area or dynamically in the execution stack depending on the semantics of the language and conventions set up by the translator. Relative positions (offsets)

in memory of various names are recorded in the symbol table. Order of evaluations of subroutine call parameters are determined and subroutine calling conventions are set up. The stack frame for subroutine calls are built. The output of this phase is a sequence of low level operator trees. The internal nodes of an operator tree represent low level operators and the subtrees of a node represent the operands.

Code generator: This phase takes the output of the translator and generates equivalent assembly language code for the target machine. The code generator has to generate efficient code by utilizing the target machine instruction set well and by efficient register usage. Given an input tree to code, the code generator decides the order of evaluation of the subtrees where there is a choice. The preferred order is one which minimizes register and temporary memory space usage. Usually there exists more than one sequence of instructions which computes a given tree. In such cases the code generator chooses among the possible instruction sequences. The task of selecting a code sequence from among the choices is called instruction selection. An exhaustive search to determine the best possible sequence is practically impossible. Therefore, the code generator relies on heuristic techniques to generate good quality code. Usually the particular code sequence depends on the context in which code is being generated. Hence the input tree may be restructured to reflect this context. Efficient ways to access values in memory are determined based on the addressing mechanism provided by the target machine. Another task done by the code generator is register allocation to the various values used in a computation. An attempt is made to keep the more frequently used values in registers so that the code generated runs faster. The code generator keeps track of the registers as to whether they contain useful values or are free. The register requirements of computing a tree are determined and if enough free registers are not available, registers holding useful values are made free by spilling (i.e., storing) them into memory so that enough registers become available. Once the code generator has decided upon an instruction, its assembly language representation is formatted and emitted.

1.2 The work done here

The work done here pertains to the code generation phase of the FORTRAN 90 compiler being developed. The intermediate representation of the program input to the code generator was designed. A parser driven pattern matching algorithm was used to generate a code generator automatically from a description of the machine. Specifically, the code generator does the tasks of instruction selection and register allocation. The code generator generates assembly code for the CONVEX C220 machine architecture.

1.3 Organization of this report

Chapter 1 gives an overview of the compiler being developed and the work done here. Chapter 2 discusses the various issues involved in code generation and the different methods of code generation. Chapter 3 is concerned with the design and implementation of the code generator. Chapter 4 addresses the testing of the code generator, integration of the code generator with the other phases of the compiler, compares the code generator with the existing FORTRAN compiler in the CONVEX C220 machine and concludes the work.

Chapter 2

Code Generation : Overview

Code generation is the process of mapping an intermediate representation of a program into instructions of the target machine, i.e., a code generator takes as input an intermediate representation of a program and produces as output an equivalent target program. This mapping is typically one to many. Hence, given an intermediate code we would like to generate a sequence of instructions that runs fastest and occupies the least amount of memory. Unfortunately, there does not exist a general algorithm for generating such an optimal code which can be applied to all machines. Even when they exist for special cases, they are too costly to be used in practice. Hence, practical code generators attempt to produce locally optimal code. In this chapter we consider the issues involved in the design of a code generator, and the various approaches that have been used to construct code generators.

2.1 Issues in the design of a code generator

The issues in the design of a code generator include the following:

Correctness of code: The bare minimum that a code generator must do is that it should produce correct target code for a valid input. For a real machine with a real instruction set this is not a trivial task. Ideally we would like to have a method that can be proved correct.

Quality of code produced: The code produced should be of high quality, i.e., the generated code should run fast and occupy less space. In order to achieve this, the full

instruction set of the target machine should be used well and other machine resources such as registers should be used efficiently.

Efficiency: The code generator itself should run efficiently. This implies that the algorithm used for code generation should be fast. The difficulty is that of balancing the requirement for the generation of efficient or optimal code with the time taken by the code generator to produce this code. For example, generation of optimal code for expressions in the presence of common subexpressions is NP-complete.

Input to the code generator: The input to the code generator is the intermediate representation (IR) of the source program. The IR's include three address representations such as triples and quadruples, tree representations, directed acyclic graphs (dags), and linearised forms of trees such as postfix and prefix code. The particular IR chosen usually depends on the code generation algorithm.

The target program: Given a target machine the code produced by the code generator may be absolute machine code, relocatable machine code, assembly language, or some internal forms of these used by phases following the code generator such as a peep-hole optimizer.

Memory management: Storage has to be allocated to program variables and constants, and to compiler generated intermediate values, actual parameters, return addresses, and the target code itself. Actually, this is done together by the front end and the code generator. The front end may assign relative addresses to names, map source language types to machine language types, determine their alignment, compute storage requirements of common data areas and initialise them. Storage for register spills may not be possible to be allocated before code generation. Hence this may have to be done by the code generator.

Context switching: When a subroutine is called, code must be emitted for saving the information about the caller's environment, such as registers and status information, for passing parameters and return values, and for setting up the called subroutine's environment. The target machine as well as the source language may specify conventions to be followed regarding subroutine calls and returns.

Instruction selection: For a given IR input there is usually more than one target instruction sequence that can perform the computation. The particular sequence that is chosen depends on the context in which code is being generated. For example, if a value of a variable is already in a register it need not be reloaded. This requires that the code generator maintains the context in which it is generating code such as keeping track of the contents of registers and subsequent uses of values computed. Also, the instruction set of the machine should be used effectively such as using special instructions when possible.

Register allocation: *Register allocation* is the process of deciding which values to keep in registers at some point in the program. *Register assignment* decides the particular register assigned to each value for which a register has been previously allocated. An operand in a register is usually faster to access than a memory operand and also results in shorter code. Hence the code quality can be improved by keeping the most frequently used values in registers. But, the number of available registers is limited and hence efficient utilisation of registers is important. In order to perform certain operations, some of the operands may need to be in registers. In these cases, the evaluation of an expression may require a minimum number of registers simultaneously. If the required number of registers is not currently free, registers which contain useful values have to be spilled into memory.

The difficulties in the optimal assignment of registers are:

- Optimal assignment is NP-complete in the presence of common subexpressions. This is because optimal assignment depends upon knowing the evaluation order while this in turn depends upon register allocation (see [Wulf75]).
- Almost all real machines have register usage conventions especially as regards subroutine calls and returns.
- Presence of different register classes which have similar operations on them. For example, address registers used in memory addressing and the "general purpose" registers.

In practice, there are *preplanning* and *on-the-fly* register allocation strategies. A preplanning strategy estimates the register requirements of a computation to be done

and avoids register spills by precomputing values into memory if enough registers are not available. [ASU86] gives an estimation strategy called the *sethi-ullman* computation. This has been used in the Portable C compiler (see [John77] and [John78]). The BLISS-11 compiler uses a graph coloring algorithm in conjunction with estimation to assign registers (see [Wulf75]). On-the-fly register allocation strategies allocate/deallocate registers as code is being generated. Here the register manager may track information such as the variables that are in registers, the last use of a register, registers that hold temporary values, or whether the value in a register is a constant.

Evaluation Order: In the IR, there is usually a lot of freedom in choosing the evaluation order for operations. We would like to choose an evaluation order which requires the fewest number of registers or temporary memory locations and/or one which runs fastest. Here again finding the optimal evaluation order is NP-complete in the presence of common sub-expressions (see the register allocation issue above). Heuristics like choosing an ordering which reduces the register requirements are used.

2.2 Code Generation Methods

Broadly, code generation algorithms can be classified into two classes. The first class attempts to deal with code generation in a formal and theoretical manner typically to produce optimal code with idealised instruction sets and machine models. But real machines usually do not match these formal models. Also, the optimal algorithms are usually too costly to be practical. However, these formal approaches provide insights.

The second class deals with code generation algorithms for real machines. These code generators use heuristics to generate efficient (if not optimal) code. Initial approaches to code generation for real machines were *ad-hoc*. For each kind of operator or operand in the source language, there was a collection of routines to produce a sequence of target instructions which carried out the computation. Therefore the code generator was large, unmodular, complex, not retargetable, and difficult to debug, modify and maintain. Also a new compiler had to be produced for each new combination of source language and target machine. But both languages and machines continue to proliferate, and therefore there is a need for automating the code-synthesis phase in compilers. Retargetable and automatic code generation methods are aimed at this. We discuss these in the subsequent sections.

2.2.1 Retargetable code generation

Retargeting a compiler means to make an existing compiler generate code for a different target machine or another source language by making relevant changes in the existing compiler. A compiler is retargetable if these changes are minimal and easy. A retargetable compiler may thus be changed to produce a cross-compiler which runs on one machine but generates code for a new machine. Now, a compiler which runs on the new machine and produces code for it can be developed with much less effort using a *bootstrapping* technique (see [ASU86]).

Retargetable code generation methods can be classified into three categories: *interpretative code generation*, *pattern matched code generation* and *table driven code generation*. We briefly discuss the first two approaches in the next two subsections. Table driven code generation is discussed in detail in section 2.3. [GFH82] gives a survey of retargetable code generation methods.

2.2.2 Interpretative code generation

These methods divide the compiler into a *front-end* and a *backend*. The front-end (or *analysis phase*) analyses the source program and produces code for a *virtual machine*. The back end (or *synthesis phase*) expands this intermediate code into target code. Ideally, if the same virtual machine description is used for all machines, to create L * M compilers for L languages on M machines, only L front-ends and M backends need to be created. But this has not been possible in practice.

The code generator is basically an *interpreter* that implements a one-to-one or one-to-many mapping between the virtual machine instructions and the real machine instructions. Thus code generation is similar to macro expansion. A naive macro expansion would produce poor quality code. For example, the code generated will contain redundant loads and stores. Hence, the context in which code is being generated is maintained and used while expanding a virtual machine instruction to produce better quality code. Special code generation languages were also developed and the code generators were written in these languages in order to make the task less tedious.

This method is an example of a *procedural code generation* method. Here the implementor specifies even the low-level decisions about the kind of code to be generated in each

context. The main limitations of this method are

- Creating a common virtual machine is difficult.
- Code generation languages are closely tied to a specific language or machine.
- The code generation algorithm and the machine description are intermixed.
- The implementor has to make all low-level decisions about what kind of code is to be generated.
- Context dependent information is hard to incorporate.
- The interpreters have to be usually hand-coded.

2.2.3 Pattern matched code generation

These approaches attempt to separate the code generation algorithm from the target machine descriptions. Hence, potentially one code generation algorithm could be used for all machines. The input to the code generator is a sequence of trees. The internal nodes of a tree represent operators and the leaves represent the operands. Target machine instructions and addressing modes are encoded in the form of a library of *tree rewriting rules.* Each tree-rewriting rule consists of a subtree template that specifies a computation, a code fragment specifying an action and a specification as to how to rewrite the input tree if this rule is applied. The rule says that if there is a subtree in the input that matches the template, the action may be executed and the tree rewritten as specified. The action typically emits instructions to compute the subtree that matched. The nodes in the template tree may contain attributes and semantic restrictions.

The code generation algorithm is basically a tree pattern matcher. The pattern matcher matches the template trees to the subtrees of the input tree. In order for a subtree to match a template, the semantic restrictions specified in the template should also be satisfied by the subtree. The sequence of machine instructions emitted constitute the code for the input tree. Multiple pattern matches are usually resolved heuristically in favour of one which results in less costly code. If no pattern matches a subtree of the input, the subtree is transformed using default or machine-dependent rules and again matched. This can potentially lead to an infinite chain of transformations.

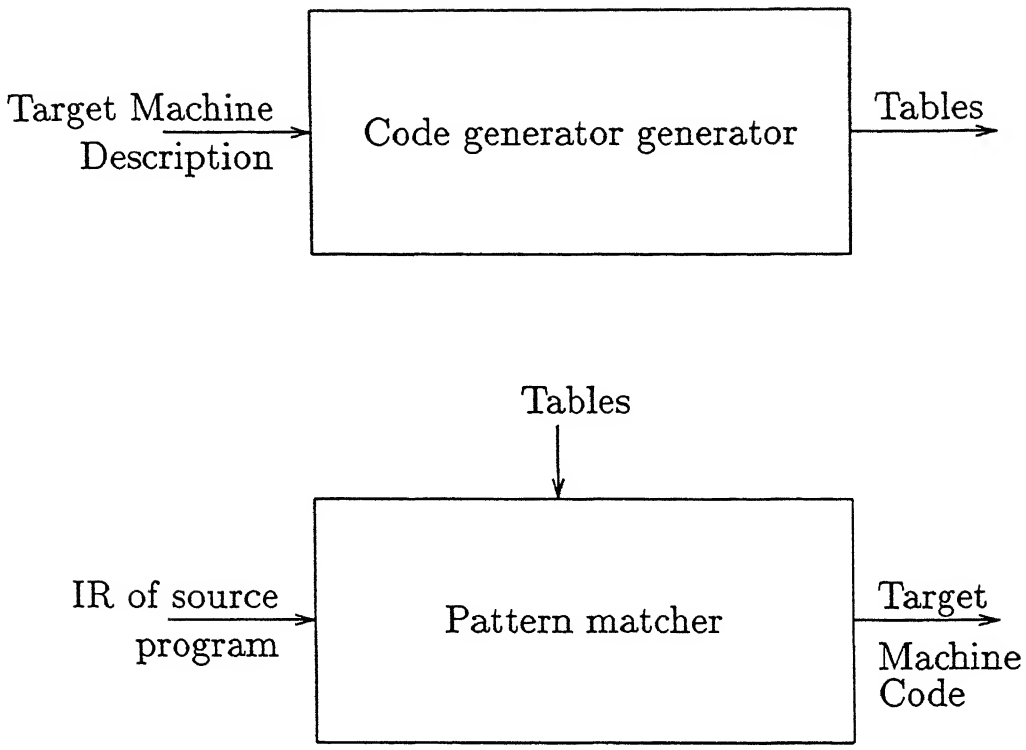


Figure 2.1: Table driven code generation

Aho and Johnson consider exhaustive search in conjunction with dynamic programming to find matches that yield optimal code sequences for expression trees (see [ASU86] for details of the algorithm). The portable C compiler developed by Johnson (see [John77] and [John78]) uses a pattern matched code generator. Section 2.4.1 gives a description of the compiler.

2.3 Table Driven code generation

The table driven approaches are automated enhancements of the pattern matched approach to code generation. These methods generate a code generator by an automatic analysis of a formal description of the target machine. In order to achieve this automation, the code generation problem is usually broken into smaller parts such as register allocation, storage allocation and instruction selection.

Figure 2.1 shows the table driven approach. The *target machine description* is a formal specification of the target machine. This is fed to the *code generator generator.* The *table constructor* in the code generator generator produces template tables automatically

from the target machine description. The input to the code generator is the intermediate representation (IR) of the program being compiled. The code generator is a pattern matcher that uses these tables to generate code by the pattern matching process described in section 2.2.3. The pattern matcher is also automatically derived by the code generator generator. The target machine description is written once for each target machine. Similarly, the code generator generator is run once on the description.

The next subsection describes a parser based table driven approach called the Graham-Glanville method.

2.3.1 Graham-Glanville method

The Graham-Glanville code generator has the same structure shown in figure 2.1.

The intermediate representation (IR) consists of a sequence of prefix linearised forms of low level operator trees. The IR is machine dependent. The IR must be constructed from the set of terminal symbols used to describe the *instruction set grammar*. The symbols may be *operators* or *operands*. An operator may be a *root level* operator that does not appear in an operand to another operator, or an *internal* operator that appears only as an operand to another operator. For example, store, jump, call are root-level operators while all arithmetic operators are internal operators. The symbols may have associated semantic attributes describing them. Control structures and procedures and function calls are also expressed as prefix expressions. High level control structures are broken down to a sequence of low level test and jump operations. Storage allocation is done prior to code generation. The address of a variable is represented by an expression describing a way to compute its actual address at execution time. Explicit dereferencing operators are used in the IR to access a value in an address. Type conversion operators are explicit in the IR.

The instructions are described by attributed productions in a context free grammar. The grammar has non-terminals for each class of registers and for condition codes. The terminal symbols are the symbols in the IR. Each grammar production in the target machine description describes one instruction in the target machine with one combination of addressing modes for the operand(s). The right hand side of the production is a prefix expression composed of grammar symbols specifying what the instruction computes. The left hand side non-terminal specifies the location of the result of the computation. The result location is a register, condition code or it may be non-existent. Commutative operations

are described with two productions each, the second one with the position of the operands swapped. The assembly information associated with the production specifies the code to be emitted to evaluate the instruction pattern. *Semantic restrictions* such as the number of a register or relationship between source and destination operands in the instructions are described by adding semantic qualifiers to the grammar symbols in the machine description.

An example of a machine description corresponding to an add instruction is:

$$reg.1 \quad \rightarrow \quad add\ const\ reg.1 \qquad "add\ const,\ r1" \qquad (2.1)$$

The meaning is to add the constant **const** to the contents of a register **reg**. The source and destination registers are the same. This is indicated by the 1 following each **reg** symbol. The instruction "add const r1" is emitted when this production is reduced.

The code generation algorithm is the standard LR(1) parsing algorithm (see [ASU86]) augmented to handle semantic restrictions and conflicts (see [Glan77]). A *reduce/reduce* conflict corresponds to a choice in the instruction to emit. The conflict is resolved by a simple heuristic that orders the rules into a best instruction first sequence according to some cost measure and chooses the cheapest among them. In the case of a *shift/reduce* conflict, a shift is chosen. This corresponds to the *maximal-munch method* (see [Catt80]) in which the longest possible pattern in the input is matched choosing a more complex instruction to emit. These heuristics do not guarantee optimal code. When a reduce operation is about to take place, registers are assigned to the source and destination register operands (if any). An on-the-fly register allocation strategy is used and the contents of registers are tracked during code generation.

The parser is said to *loop* if it makes an unbounded number of moves without reading any further input. Glanville showed that the code generator can loop only by reducing chain rules. A *chain* rule is of the form $x \rightarrow y$ where y is a single nonterminal of the grammar. [Glan77] gives a method to detect potential looping configuration by an analysis of the parsing tables using the transitive closure of a relation characterising parser moves. See [Glan77] for the details of this algorithm.

The code generator is said to *block* when it performs an *error* action for a valid input. Glanville established *uniformity* conditions on the grammar so that the code generator cannot block. The basic idea behind the uniformity criterion is that operands to an operator are valid independent of the context in which they appear. [Glan77] gives an algorithm to

check uniformity of the grammar. If the grammar is not uniform, it implies that certain operands to certain operators can occur in some contexts but not in others. In such a case the implementor has to identify these situations and ensure that these restrictions are applied in the IR as well.

A *semantic block* occurs in the code generator when it is performing a reduction, but each possible rule in the reduce set contains a semantic restriction that is not satisfied. The solution here is to provide a *default list* of shorter instructions that are equivalent to the semantically restricted instruction.

The parsing tables used in the code generator are built automatically from the machine description grammar specification. The method is almost identical to the algorithm for automatically building LALR(1) (see [ASU86]) parse tables. Loop detection and removal, block detection and default list construction for semantic blocks are also automatically done by the code generator generator. The relevant algorithms are detailed in [Glan77].

2.3.2 Comments on the Graham-Glanville method

The method is very efficient. This is because the parser does not backtrack and hence a linear time algorithm is obtained. The method can be proved to be correct (see [Glan77] for a proof). However it has the following drawbacks:

- Since the IR is low level and machine specific, it is not portable. The IR has to redesigned to retarget the code generator.
- Storage allocation is done before code generation by a machine dependent translator.
- The code generator does not maintain global context while generating code.
- Heuristic resolution of matches does not guarantee optimal code.
- It is not possible to track more than one result of an instruction such as autoincrement addressing. This problem may be solved by first generating code with only simple instructions. A post code generation compaction phase compacts these simple instructions into more complex instructions. See [KM86] for an automatic method which generates the code generator and the reduced instruction set of the machine.
- A purely syntactic approach to code generation yields a very large grammar (typically having over a 1000 productions). However, *factoring* of the grammar may be done to

reduce the size (see [GHS82] and [AGH84]).

- Treatment of complex semantic restrictions may need hand-coding. [GFH82] present handling semantic attributes with attribute grammars.

2.4 A look at some compilers

In the following sub-sections we look at three significant compiler implementations.

2.4.1 Portable C compiler

The goal in the development of this compiler was to create a portable, reliable C compiler that produced reasonably good code. [John77] gives a detailed description of the design and implementation of this compiler. [John78] discusses the theoretical aspects that influenced the design of the compiler.

The compiler consists of two passes apart from a preprocessor. The first pass does lexical analysis, syntax analysis, symbol table maintenance, type checking and the introduction of coercion operators. Machine dependent aspects such as initialization, code for subroutine prolog and epilog, storage allocation are also done. The output of this pass is the intermediate representation (IR) of the program which consists of a mix of decorated trees and assembly code for initialization and storage allocation directives.

The second pass generates assembly code for the IR. Each tree has a set of goals (called a *cookie*) that would be acceptable, such as whether the tree is to be computed for its side effect or the value, whether to store the result in memory or a temporary register. The code generator consists of two subphases. The first subphase estimates resource requirements (registers) using sethi-ullman computation and identifies subexpressions to be pre-computed and stored in temporaries. The second subphase first generates code to compute and store these subtrees. and then generates code for the tree. It is an error for the second phase to run out of registers. This is used as a correctness check. Actual code generation proceeds by tree rewriting as described in section 2.2.3. A "maximal munch" approach (see [Catt80]) is used to resolve multiple matches. If no template matches the tree, various heuristic techniques such as modifying the cookie, matching a subgoal of the computation, or applying default rewriting rules on the tree are tried.

The compiler was ported to a large number of machines by manual rewriting of the machine dependent portions of the code. Rewriting of the templates used by the code generator and the tuning of the ported compiler are not easy. If there are not sufficient number of templates to cover all cases, the code generator can block or loop (by applying various rewriting rules) for a valid IR input. It is not possible to ensure that there are a sufficient number of templates. Portability, and not efficiency of code produced, was the primary goal of the compiler and this was achieved.

2.4.2 BLISS-11 compiler

The BLISS-11 compiler is an optimising compiler for the PDP-11 target machine. The main goal in the design of the compiler was the production of extremely efficient object code. The optimisations were aimed more in saving memory than time. [Wulf75] gives an elaborate account of the design and implementation of the compiler. The phases involved are touched upon below.

The phase LEXSYNFLO performs lexical, syntax and flow analysis and produces a tree representation of the program decorated with information necessary for future optimisation. The DELAY phase shapes and optimises the tree, and determines the order of evaluation. Register requirements for each node is determined based on the sethi-ullman computation suitably altered to handle common subexpressions. The TNBIND phase allocates memory and registers to temporaries created in the previous phases. A TARGET subphase identifies classes of temporaries that may preferably be packed into the same physical location. The frequently used temporaries which may be allocated registers are identified by the RANK phase. The PACK phase does the actual allocation of registers and memory.

The CODE phase does an execution walk of the tree and generates code to evaluate each node. The code generation approach used is relatively ad hoc. A great deal of tedious special case analysis is done which critically ensures good quality code. The FINAL phase performs a collection of ad hoc peephole optimizations on the code produced by the CODE phase and then emits the final relocatable code.

2.4.3 PQCC project

The goal of the PQCC (Production Quality Compiler Compiler) project was to build an automatic table-driven compiler writing system. [Wulf80] gives an overview of the project

and [NSP86] gives a retrospection on the same. The Bliss-11 compiler (see [Wulf75]) was used as a basis for the compiler and the quality of its output code. The basic strategy in developing the compiler was to formalize and parametrize standard techniques so that they could be automated.

The compiler is divided into many phases, each with an associated phase generator which generates that phase automatically. Tables supply source language and target machine information relevant to each phase. The output of the front-end of the compiler is an abstract syntax tree. The FLOWAN phase does data and control flow analysis and identifies feasible optimisations. The DELAY group of phases transform the tree by applying desirable optimisations, decides upon the order of evaluation and shapes the tree. Addressing modes for operands are determined. The TNBIND phase does a pseudo code selection to allocate registers and temporaries. The packing subphase of this phase allocates the most frequently used values to registers. Attempt is made to allocate the same memory locations to temporaries based on preference information. A table driven heuristic pattern matching method using a maximal-munch approach (see [Catt80]) to resolve multiple matches is used to generate code. The tree templates are automatically derived from a machine description . Instructions with multiple results are described using a template for each result. The FINAL phase performs peep-hole optimizations.

Chapter 3

Design and implementation

In this chapter, we describe the design and implementation of the code generation phases of the FORTRAN 90 vectorizing compiler. A parser based table driven automatic code generation method was used which is based on the Graham-Glanville technique described in section 2.3.1. The design of the code generator was based on similar experiments described in [AGH84] and [GHS82]. Since a parser generator specifically designed to serve as a code generator generator was not available, the parser generator, *bison*[1] was used instead as the table generator.

3.1 The code generator structure

This section describes the structure of the code generator. Figure 3.1 shows the logical phases involved in the code generator. With respect to the overall structure shown in figure 1.1 the output of the translator is the input to the code generator. The output of the translator is a low level operator tree. Memory allocation for the various names in the program is assumed to be done by the translator. There are three main phases in the code generator. They are:

Transformation phase: The transformation phase *restructures* or *shapes* the tree output of the translator. Typically, operators that are not supported by the IR input to the pattern matcher are replaced and the tree is *canonicalized*. Certain reorderings may

[1] *Bison* is identical to the parser generator *yacc* in terms of input syntax. See [John75] for details regarding *yacc*

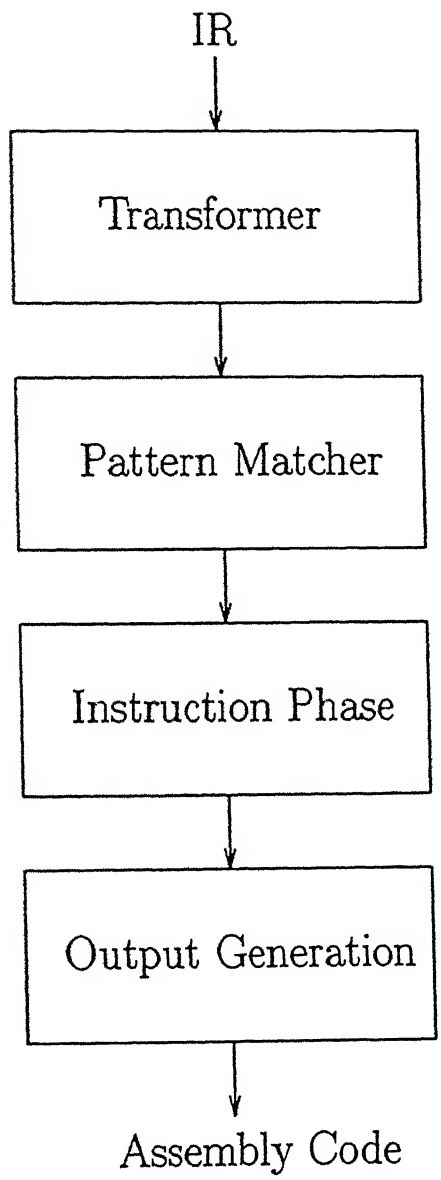


Figure 3.1: The logical structure of the code generator

be done to improve the quality of the code produced. The output of the transformer is a prefix linearized form of the tree.

The pattern matching phase: The output of the transformer is fed to the parser based pattern matcher. The pattern matcher uses the Graham-Glanville parsing technique described in section 2.3.1. The parser is driven by tables generated by *bison* from a target machine description grammar. The parser selects instruction patterns specified by the target machine grammar and gives it to the instruction phase.

Instruction phase: The selected instruction pattern fed to the instruction phase typically consists of an operator and its associated operand(s). Information in the form of attributes is also supplied along with the operators and operands. The instruction phase may generate zero or more assembler instructions. This phase consists of two subphases, namely, instruction selection and register allocation. The instruction selection subphase checks the semantic restrictions, if any, associated with the selected syntactic pattern and does the final selection for the instruction(s) to be emitted. After the instructions have been selected the register allocator is called to allocate registers required for the operands and the result of the selected instructions. Register allocation is done *on-the-fly*.

Output generation: This phase does the formatting of the instructions in the assembly language notation of the machine, and prints the instructions generated.

In the rest of the chapter we shall discuss in detail the design of the IR and the various phases outlined above. But before that we briefly describe the target machine architecture and its assembly language in the next section.

3.2 The target machine and language

Since the compiler being developed is not a parallelizing compiler, only the uniprocessing capabilities of the machine are used and hence we consider only these here. Also, all privileged instructions were excluded.

The machine architecture provides three classes of registers:

- Address registers named A0 through A7. Out of these, A0 is the stack pointer register SP, A6 the argument pointer register AP, and A7 the frame pointer register FP.

- Scalar registers numbered S0 through S7. S0 is used for return values of sub-routines.
- Vector registers V0 through V7. Each vector register has a length of 128. Each element in the vector is 64 bits long.
- Various special purpose registers for vector processing such as VL, VS, VM are used to specify the vector length, vector stride (i.e., size of each vector element), vector mask to select a subset of vector elements in a vector, for an operation.

The instructions provided include the following:

- Memory reference instructions, such as load (ld) and store (st). The addresses of memory locations can be immediate, absolute or indexed with one optional level of indirection.
- Arithmetic and logic instructions such as add, sub, mul, div, or, and, xor, compare (cmp).
- Data conversion instructions.
- Vector-reduction instructions (sum, prod, max, min, all and any).
- Data transfer instructions between registers.
- Instructions to set the various special purpose registers, and
- Program control instructions such as for subroutine call and return (call and ret) and conditional and unconditional jumps (jbr, jbrs, jbra).

Operations can be broadly divided into two groups– *scalar* operations and *vector* operations. Scalar operations operate with scalar operands while vector operations operate with at least one vector operand. *All operators except load and store operate on register operands only.* In general, operators can operate on data of sizes byte (8 bits), halfword (2 bytes), word (4 bytes), long (8 bytes), single precision floating point (4 bytes), and double precision floating point (8 bytes). In general, each operator is qualified by the type on which it operates. For example, the addition operator for words is **add.w** and that for longs is **add.l**. Most of the vector operators can have an optional mask which is either true (t) or false (f). Thus we can have vector add operations **add.w.t** and **add.w.f**. The special register VM contains the mask bits.

CONVEX C220 assembly language was chosen as the target language for the code generator. The assembler provides a powerful language with the following facilities.

- There are pseudo-ops to specify data and code segments. Data and code segments can be generated in any order. The assembler automatically collects them together. Also, there are pseudo-ops to declare and initialize data areas and common areas. The assembler detects conflicting initializations of common areas.
- Span-independent jump instructions have been provided. The assembler does automatic resolution of jump instructions having a short span into branch instructions.
- Allows symbolic labels to be used anywhere in the program. The symbolic labels may refer to locations in the program or they may refer to constant expressions. Labels may be referred to prior to their definition.
- Provides some convenient addressing modes not directly supported by the machine's instruction set. An example is the absolute addressing mode.

For a detailed description of the CONVEX C220 architecture and assembly language see the architecture manual [ConArc] and the assembly language manual [ConAs] for the CONVEX C220 machine.

3.3 Intermediate representation

The intermediate representation (IR) was designed by looking at the target language alone. The IR defines the input to the pattern matcher. The IR of a translated source program consists of a sequence of operator trees in their prefix linearized form. The IR assumes that memory allocation has been done by the translator. Each tree may be optionally labelled by a name. This is specified using the **label** operator, which takes a name and a tree as operands and associates the name with the address of the location of the code for the tree. Structurally, the IR consists of a sequence of subroutines. The beginning of each subroutine is specified in the IR by attaching the name of the subroutine as a label to the first statement of the subroutine. The return instruction may optionally specify a subtree to be computed and returned.

Subroutine calling conventions of the machine have to be followed during subroutine call/return. Parameters to a call should be evaluated and pushed into the run-time stack

before control is given to the subroutine. The return value of a subroutine is returned in the scalar register s0, and hence should fit into this register. This return value may be accessed by the name "RETVAL" in the IR. Hence this name should not be used for any other variable in the program. Details regarding subroutine calling conventions may be found in [ConAs]. Every return statement has to be followed by a labelled tree or it should be the end of input.

3.3.1 Operators in the IR

Broadly, the operators have been divided into *root-level* operators and *internal* operators, as in the Graham-Glanville code generator. The root-level operators include the *load* (ld), *store* (st) , *compare* (lt, le, eq) operators and *jump* (jmp) and *call* (calls, callq, call) instructions. These are executed for their side-effect only, such as changing a value in memory, setting a condition code, or altering program flow. The internal operators return values in registers and include the various vector and scalar instructions for arithmetic and logical operations.

In general, the operators are *typed* as *byte, halfword, word, long, single-precision floating point, or double precision floating point.* The operators are represented by the basic operation followed by a '.' and a letter showing the type (b, h, w, l, s, d respectively for the above types). Some operators such as the logical operators do not have a type specified. In these cases the size of the operand registers implicitly determine the type. The operators provided in the IR closely follow those in the instruction set of the machine. For example, the machine provides only three relational operators, namely *less than, less than or equal to and equal to.* Hence the IR also provides only these three comparisons. Other relational operations have to be expressed in terms of these. This should be done by the translator or the transformer before the pattern matcher is invoked. However, there are some IR operations that do not have a direct operation on the machine. These *pseudo-operations* may emit more than one instruction. For example, the machine does not provide type conversion operators between all pairs of types; the IR does. Also, suitable tests are assumed to be conducted immediately before a conditional jump.

Operations to load a value into general purpose registers are not explicitly reflected in the IR. However, it is possible to set, as well as access, the values of the special purpose registers such as SP, FP, AP, VM, VL, VS, explicitly in the IR. Instructions to load the operands into registers is emitted at appropriate places by the register manager during the

instruction phase. This is because register allocation is done within the code generator. Like the load operators, the register move operators for the general purpose registers are also not provided in the IR. The move instructions are also emitted by the register manager when the value in a register has a future use and hence needs to be preserved.

It is possible to stop the code generator at fixed points in the IR using the **break** input symbol. This can be used to inspect the status of the registers, the variables that are in registers, their use counts and the code generated so far. Use counts for variables may also be specified at this point. When interrupted like this, the code generator displays a menu of these commands and prompts the user for commands.

3.3.2 Operands in the IR

Valid operands to an operator may be constants or names representing values in memory locations. The IR is uniform as explained in section 2.3.1. The operands have attributes describing them. For example, the symbol **const** (for a constant value as an operand) has as its attribute a string representing its value as well as a type. The attribute is described by a ':' after the symbol and preceding the attribute value. For example, **const.w:** 1000 specifies a 32 bit constant of value 1000. A value in memory is specified by a subtree representing the address computation involved explicitly. The root of the subtree should be the dereferencing operator **ind** (**vecind** for accessing vector operands in memory). The offset of a name in its area in memory (such as static data area or run-time stack) is specified by the IR symbol **addr** with the name string as an attribute. An example is **addr:** x for the offset of name x in its memory area. Addresses of memory locations are implicitly 32 bits long, i.e., of type 'w'.

Operands to an operator should be of the same type as the operator except for the indirection and data conversion operators. Necessary conversion operators have to be inserted in the IR tree, before it is input to the pattern matcher. The code generator assumes this and hence does not track the types of operands internally.

If a register is allocated before pattern matching, its value may be specified in the IR using the three symbols, **Areg**, **Sreg** and **Vreg** depending upon whether the class of the register is address, scalar or vector. The number of the register allocated is specified as an attribute of the symbol. The register allocation routines used in the instruction phase may be used to do this allocation before pattern matching.

3.3.3 Other inputs to the code generator

Apart from the IR operator trees, the code generator is given two more inputs. The first is a file containing symbol table information regarding the various names used in the IR. The information in the symbol table entry corresponding to each name specifies whether the name is allocated storage in the static storage area or in the run-time stack, and the offset within that area. The data type of the name is also stored. A second input is a file containing use count information for the names. The use count of the lvalue as well as rvalue may be specified in the file in a particular format. This second input is optional. In case no use counts are specified, the default use count of 1 is used for each name. The file names are input to the code generator as command line arguments when the code generator is invoked. A small program has been written to create the symbol table file in the proper format. It prompts the user for the necessary information and reads the information from the standard input.

3.4 Target machine description

In this section we discuss the design of the target machine description grammar. This grammar describes the instruction set of the machine. This is the input to the parser generator *bison*.

3.4.1 Representation of information

One of the crucial issues in the design of the target machine grammar was to decide how much of the information supplied to the code generator was to be in the form of *syntax* (i.e., through the machine grammar) and through *semantics* (as attributes of various grammar symbols). There exist two approaches to represent information – the *purely syntactic approach* and the *hybrid approach*. See [AGH84] for a detailed discussion of these two approaches and the tradeoffs involved. In a purely syntactic method the attributes of a symbol are encoded into that symbol by creating a *family* of new symbols. For example, to encode the *type* of an **add** operator, we could create separate symbols like **addb, addh, addw, addl, adds, and addd**. This process creates a family of productions for each production in a grammar where the type is kept as a semantic attribute of **add**. This encoding process is called *type crossing*. In the *hybrid syntax-semantics* approach, the symbol might be

declared by the single symbol **add** along with a field which stores the type. As is obvious, the syntactic approach increases the size of the grammar combinatorially.

The hybrid approach allows a small machine grammar and parse tables, but the semantic actions corresponding to a production have to check the attributes explicitly and decide the parsing actions. The parser may loop or block based on both syntactic and semantic attributes. Hence automatic methods to detect and clear blocks, as can be done in a purely syntactic method, (like in the Graham-Glanville method: see section 2.3.1) cannot be used. Also, the information describing the machine is not confined to the grammar alone, it is in the semantic attributes also. But this may affect only the ease with which the code generator might be retargeted. Ensuring correctness of the syntactic method is easier. Despite these advantages of the syntactic approach, the size of the parser tables and the combinatorial increase in the number of productions is a serious disadvantage.

We have adopted a hybrid approach. The subset of the instruction set for which we were generating a code generator had about 75 basic opcodes. After type replication and counting instructions with different class of registers as separate, there were over 800 instructions. This does not include the replication over the various addressing modes for memory reference instructions. Hence a *flat* grammar (as is in Glanville's method described in section 2.3.1) was impractical. Hence the machine description grammar was *factored* according to the structure of the instruction set (see [AGH84] and [GHS82]). Not all productions in the factored grammar emit instructions. Some of them encapsulate information regarding the right hand side of the production into a *signature* associated with the left hand side nonterminal. Others are for parsing purposes.

3.4.2 Factoring of the grammar

To begin with all addressing modes were factored into the single non-terminal **effa**. The orthogonal nature of the instruction set prompted us to make the following factorings in the grammar. The first was to track the *type* of an operator as a semantic attribute. Hence there is only one terminal symbol for each operation with different types. This semantic tracking of types considerably reduced the size of the grammar in terms of the number of productions. Also, the *mask* field of the vector operations were tracked as an attribute which could have values true ('t') or false ('f').

Another factoring grouped together all the operators (without considering their types)

having the same *operand configurations* into disjoint sets. An operand configuration for an instruction is the kind (such as address, scalar or vector register, constant, or an effective address) of each operand. The sets have to be disjoint so that reduce/reduce conflicts are avoided. Each such operator set was represented by a new non-terminal. An example is the grouping together of the operators **and, div, mul, or, shf, sub, xor**. This factoring was systematically done by first classifying the operators according to their arities. Thus all the binary operators were grouped together as also the unary operators. These groups were examined to see if two operators in a group had different operand configurations, and if so the group was split into two or more groups. As an example, initially the binary operator group contained the **add** and the **sub** operations (among others). But it was found that the **add** instruction operated on the operand configuration with a scalar register and an address register while the subtract operation has both the operands of the same register class. Thus the binary operator group was split into two groups, one containing **add** and the other the **sub** and other operations. The final groups obtained after a series of splits resulted in **add** being in a group of its own and **sub, mul, or, shf, xor, and, div** in another group.

In order to make possible more groupings of operators into disjoint sets, operations valid for vector and scalar operations such as **add, sub, neg** (negate), were represented with two tokens each, one for the scalar operation and the other for the vector operation. For example, addition has two terminal symbols, **add** and **vecadd**. It was found that the groupings possible for the scalar operations and the vector operations were different. Therefore, if separate operators are used, the grammar could be factored more. The sets have to be disjoint as there will be reduce/reduce conflicts otherwise. For example, when the parser sees the **add** token, it cannot decide which of the nonterminal representing the sets of which **add** is a member, to reduce the token. Introduction of the **vecadd** token removes this conflict. This made possible the grouping of operations **vecadd, vecand, mask, merg, vecmul, vecor, vecxor** into a single set. But as we saw earlier in this section, the scalar **add** operation is in a set of its own. We did not have a set of scalar operators and a vector operator set which could be obtained by simply replacing each scalar operator with its corresponding vector operator. Hence none of the vector operators thus introduced were redundant, i.e., they were essential for the grouping to be done.

Nonterminals were introduced in the grammar for each class of registers. This was done

because all the instructions have only register operands and a single nonterminal resulted in an overfactored grammar with too few productions. Also the parser would select too many instructions that are not in the instruction set and this would lead to many default instruction sequences. With the introduction of these nonterminals, the operators that were commutative but having both scalar and address register class operands were described with an additional production with the operand positions swapped.

3.4.3 The vecind operator

In order to avoid unresolvable reduce/reduce conflicts two dereferencing operators **ind** and **vecind** had to be introduced. This conflict arose because certain vector operations operate with scalar as well as vector register operands. Therefore, when a dereferencing operator is seen in the input, the parser cannot decide whether to treat the operand as a scalar or a vector operand. An example of such a conflict is the two instructions

$$add.w\ V_i,\ S_j,\ V_k \quad and$$

$$add.w\ V_i,\ V_j,\ V_k$$

When the input corresponding to the second operand is seen, the parser does not have enough syntactic information to decide whether to load the operand into the scalar or vector register.

3.4.4 Syntactic blocks

The IR is uniform, i.e., valid operands to an operator and the meaning of the operator depend only on the operator and not on its context. Hence the grammar developed above was checked to see whether the parser blocked on a valid IR input. It was found that the the only way blocking could occur on valid input was when the parser incorrectly loaded a value into an address register when the operation needs the value in a scalar register or vice-versa. This possibility was removed by providing grammar productions to move values between these register classes.

3.4.5 Conflict resolution

The "maximal munch" ([Catt80]) approach was used to resolve the shift/reduce and reduce/reduce conflicts in the parser. Thus in the case of a shift/reduce conflict the parser

because all the instructions have only register operands and a single nonterminal resulted in an overfactored grammar with too few productions. Also the parser would select too many instructions that are not in the instruction set and this would lead to many default instruction sequences. With the introduction of these nonterminals, the operators that were commutative but having both scalar and address register class operands were described with an additional production with the operand positions swapped.

3.4.3 The vecind operator

In order to avoid unresolvable reduce/reduce conflicts two dereferencing operators **ind** and **vecind** had to be introduced. This conflict arose because certain vector operations operate with scalar as well as vector register operands. Therefore, when a dereferencing operator is seen in the input, the parser cannot decide whether to treat the operand as a scalar or a vector operand. An example of such a conflict is the two instructions

$$add.w\ V_i,\ S_j,\ V_k \quad and$$

$$add.w\ V_i,\ V_j,\ V_k$$

When the input corresponding to the second operand is seen, the parser does not have enough syntactic information to decide whether to load the operand into the scalar or vector register.

3.4.4 Syntactic blocks

The IR is uniform, i.e., valid operands to an operator and the meaning of the operator depend only on the operator and not on its context. Hence the grammar developed above was checked to see whether the parser blocked on a valid IR input. It was found that the the only way blocking could occur on valid input was when the parser incorrectly loaded a value into an address register when the operation needs the value in a scalar register or vice-versa. This possibility was removed by providing grammar productions to move values between these register classes.

3.4.5 Conflict resolution

The "maximal munch" ([Catt80]) approach was used to resolve the shift/reduce and reduce/reduce conflicts in the parser. Thus in the case of a shift/reduce conflict the parser

shifts. This helps in the choosing of the special case instructions like the add instruction with an immediate constant, than a register add which requires an extra load instruction to load the constant into a register. In the case of a reduce/reduce conflict the parser chooses the longer or the less costly production. This results in the use of more complex addressing modes and hence fewer instructions. The productions are arranged so that the longer ones come first in the grammar description. In the case of equally long productions, the one with the less costly instruction is put first. Thus the cost of the instructions is not explicitly represented, rather, it is implicit in the machine description.

One important group of reduce/reduce conflicts were the productions which loaded variables and constants into registers. In the machine many scalar operations can be done by loading the operands into either address registers or scalar registers. The parser does not have enough information to decide syntactically the class of the register to load a value. Hence, as a heuristic, all rvalues of variables are loaded into scalar registers and lvalues and address constants are loaded into address registers. Therefore, register moves may be emitted in case the heuristic loaded a value into the wrong class of register. This can be solved only by allocating registers before pattern matching as explained in section 3.3

3.4.6 Looping

As we discussed in section 2.3.1, the parser can loop only by *chain rule reductions*. There are two productions in our grammar which form a loop. The productions are:[2]

$$areg \rightarrow sreg$$

$$sreg \rightarrow areg$$

These productions cause a loop only if they are chosen for reduction over other longer reductions, in a reduce/reduce conflict. Hence this looping configuration is trivially removed by the conflict resolution rules discussed in section 3.4.5

3.5 Instruction phase

This phase is invoked in the form of semantic actions to be performed when the parser has seen the right hand side of a production and is about to reduce by that production.

[2]The nonterminal areg and sreg stand for the address register and scalar register respectively

The semantic actions are executed just before the reduction is carried out. The attributes associated with the right-hand side non-terminal are synthesized. The subphases involved in this phase are instruction selection and register allocation. We discuss the two subphases below.

3.5.1 Instruction selection

Instruction selection is driven by the syntactic pattern selected by the parser. Since the types of operators are not represented in the grammar, the type of the operator is checked to see whether the operand configuration selected by the parser has a direct instruction in the machine's instruction set. This is done by functions written specific to each operator and operand configuration. The address of the function to execute is obtained by searching an *instruction table* with the operator and the operand configuration as indexes. The structure of the instruction table is described in section 3.5.3. The routine checks to see if the particular type of the operand has a direct instruction with the particular operand configuration. Below we discuss the case when there is a direct instruction. The case where the instruction does not have a direct instruction is discussed in section 3.5.2 under semantic blocks.

If the instruction pattern selected has a direct instruction, the register manager is called to assign registers to the source and destination register operands. Attempt is made to reuse the source operand register for the destination operand register. If one of the source operand register itself is the destination operand (as in the case of most of the scalar instructions), that register has to be used as the destination operand. But if the source register value has a next use the following actions are done.

If one of the register operands do not have a next use and the operator is binary and commutative, then the operands are swapped provided the register classes of both the operands are the same[3]. The latter condition is imposed because the instruction selector cannot change the destination register class specified by the syntactic pattern. The information, whether an operator is commutative or not, is obtained from the instruction table.

Failing above, a free register of the appropriate class is used as the destination register. For this, code is first emitted to move the source/destination operand register to this new

[3] In fact the operator is defined to be commutative in the instruction table only if the operands are of the same register class apart from being commutative in the mathematical sense.

register. Then this new register is used. If the instruction allows the specification of a destination register explicitly (as in the case of most of the vector register instructions), the move instruction is not needed.

3.5.2 Semantic blocks

Semantic blocks occurred when the particular type of an operation was not directly supported in the machine, or when the operands were not in the proper class of registers or both. Two methods were adopted to remove semantic blocks. The first method was to treat the instruction causing blocking as a pseudo-instruction and emit a sequence of instructions to implement it (see [GHS82] and [AGH84]). For example, the machine's instruction set does not provide a full set of instructions to convert between possible machine data types. The conversions not available have been implemented as pseudo-instructions. As a specific example, there is no instruction to convert a *byte* value in a scalar register to a *long* value. This was implemented by the sequence to convert the byte value to a *word*, and then, the word was converted to *long* as shown below.

$$cvtb.w\ S_i$$

$$cvtw.l\ S_i$$

In a similar manner the *byte* add instruction for address register operands was implemented by using the *word* add instruction. This was done by a sequence of instructions to convert the operands to *word*, performing the *word* addition on them and then converting the result back to *byte* type.

The second method was to move the operands into the proper class of registers so that the operation becomes possible. For example, there is no byte add operation to add a byte constant to the contents of an scalar register. This was implemented as a pseudo-operation which first loads the value into a scalar register and then performs the addition. In case both the methods were applicable, the cheaper one in terms of number of instructions and the number of extra registers required was chosen.

A pseudo-instruction may cause other pseudo-instructions to be emitted. For example, the immediate byte addition of a constant and an address register would first load the constant into an address register and then emit the sequence of pseudo-instructions for the byte add addition we discussed above.

The pseudo-instructions were implemented by writing a specific function for each operator and operand combination. This default function to be executed is obtained by searching the instruction table.

3.5.3 Instruction table

Each entry in the table has the following fields :

opcode: This gives the basic operation code like ADD or VECADD. The table is kept sorted on this field. The opcodes are those that are in the IR. This field along with the two source operand type fields are used for indexing into this table.

type fields: These are bit fields which indicate the types for which the operation is provided in the instruction set. There is a bit corresponding to each possible type in the machine. The bit is set to 1 if the operation is available on the machine and set to 0 otherwise. The types are byte (b), halfword (h), word (w), single-precision floating point (s), double-precision floating point (d), extended (x) and the null type.

mask bits: These are used for the vector operations to indicate whether a true, false or no mask can be applied to an instruction.

operand types: These two fields specify the source operand class, i.e., whether the operands are address(AD_REG), scalar (SC_REG), vector (VE_REG), or special registers, whether they are memory operands (EFF_A) or non-existent (NILL).

destination operand type: This indicates whether the result exists and if so whether it is same as one of the source operands, whether it could be a new register, or whether the result is in memory.

default function: This is a pointer to the function to be invoked to check the semantic restrictions and to emit default instruction sequences.

print string: This gives the character string corresponding to the operator in its assembly instruction.

commutativity: Indicates whether the operator is commutative with respect to the operand configuration.

The instruction table is searched using binary search with the operator and the type of the operands as the indexes into the table.

3.6 Register management

This subphase handles the allocation, assignment and deallocation of registers. The register manager implements an unbounded number of *virtual registers* on top of the *physical registers* provided by the target machine. Initially when the pattern matcher requests a register, a virtual register is allocated and its number is returned. This number is actually an index into the register descriptor maintained by the register manager. When the value in a virtual register is to be used for emitting an instruction, the register manager assigns a physical register to the virtual register and, if necessary, loads the appropriate value into the physical register. This physical register is obtained from a list of free physical registers kept. There is one list for each register class. The physical register or the location in memory, where the value of the virtual register may be found, is kept in the register descriptor. Thus, the register manager transparently handles the mapping between virtual registers and the physical registers. Register spills and unspills are also handled in a transparent way. In the following subsections we discuss the register and address descriptors, the handling of register spills, and the assignment of specific physical registers.

3.6.1 Register descriptor

The register descriptor keeps necessary information regarding the content of each virtual register of each register class. Figure 3.2 shows the structure of the register descriptor. The descriptor is implemented as an array for each class of registers. Each element in the array describes a virtual register. The fields in the descriptor give the following information.

status: This tells whether the virtual register is free or allocated; if allocated, where its value may be found, i.e., whether it is in the data area, in a physical register, whether it has been spilled and the location in stack where its value has been spilled or whether it is a constant.

location: This gives the physical register number or the address where it is stored depending upon whether the value is in a physical register or it has been spilled.

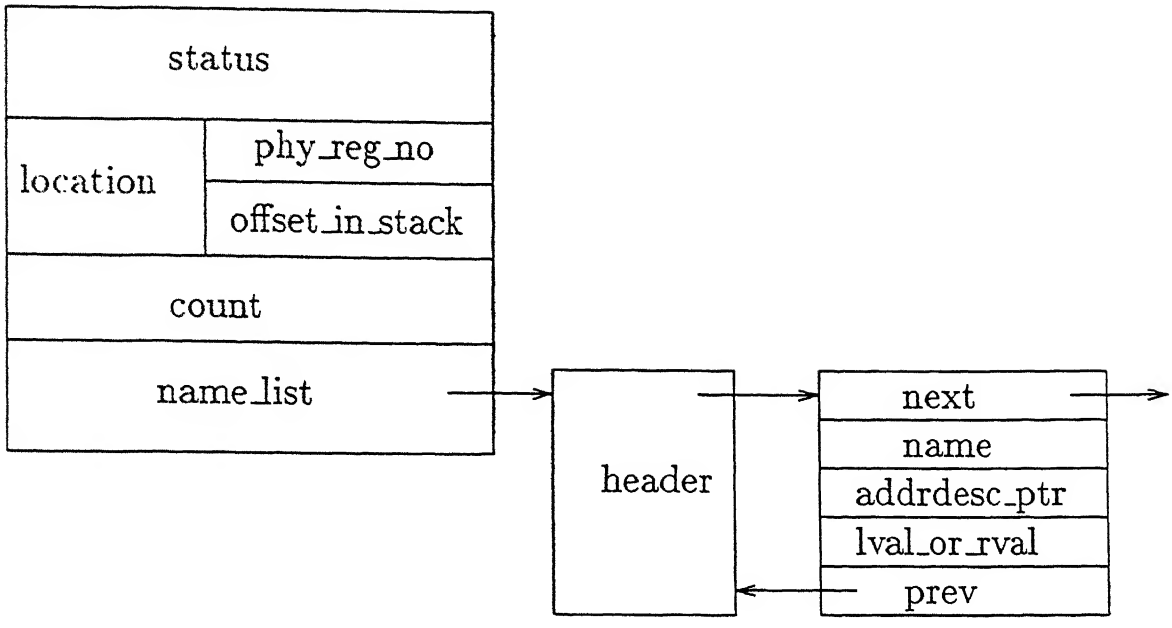


Figure 3.2: The register descriptor

count: A count of the number of names that have their value in this register.

name list: A pointer to a doubly linked list of the names that have their value in this register. This may include constants. Each element in this linked list also has a field which specifies whether the lvalue or the rvalue (or both) of the name is in the register and a pointer to the address descriptor entry corresponding to the name.

3.6.2 Address descriptor

The address descriptor keeps the following information about each name:

- The string corresponding to the name.
- The virtual register number and its class that contains its lvalue or rvalue.
- The use count of the lvalue and rvalue of the name.

The insertions leave the list in sorted order according to name. Whenever a register is requested for a name, the address descriptor is searched first to see if the name already has a virtual register of the desired class. Otherwise, a new virtual register is allocated and the address and register descriptors are updated to record this allocation.

3.6.3 Register spills

Register spills are required when the register allocator runs out of free physical registers. The register chosen to spill is the one that has been used least recently. In order to get the least recently used register, a usage stack is maintained for each register class. Whenever a physical register is used in emitting an instruction, the register is put at the top of the stack. If the register was already in the stack, then it is first removed from the stack and then pushed back on to the top. The register at the bottom of the stack will be the least recently used.

The spilling and unspilling of vector registers is very costly. A vector spill requires the saving of the VL and VS registers by pushing their values in the stack, setting VS and VL to default values, issuing a store instruction and restoring the values of VS and VL. A vector unspill requires a similar sequence of operations with the store replaced by a load. Also the size of memory required to spill a register is 1kbytes. Hence vector register spills are to be avoided as far as possible.

Once a physical register has been chosen to spill, the status of the virtual register it represents has to be updated. Hence, the virtual register corresponding to this physical register, has to be known. This information is kept in a data structure called the *physical register descriptor*. The physical register descriptor also indicates whether the physical register is free or allocated. This is needed because the virtual register information is valid only if the physical register has been allocated. If the virtual register value is also in memory, then, no instructions are emitted to store the register in memory. The status of the virtual register descriptor is updated to record that the value is only in memory and not in the physical register.

On the other hand, if the virtual register value is only in the physical register, the value is saved on the run-time stack in the space allocated to local variables in a subroutine (see calling conventions in [ConAs]). Memory of size equal to the size of the register being spilled is allocated in this space. The register manager keeps track of the stack space allocated to such register spills so that they may be reused. This is done by keeping a list of "spill area" allocated on the run-stack and later on freed when the register is unspilled. This list is kept for each register class. Whenever spill area needs to be allocated on the stack to spill a register, the corresponding "spill area list" is checked first to see if there is a vacant slot. Also, the spill areas allocated to one basic block can be reused while generating code

for another basic block. This is because a value spilled on the stack from a register has all its uses within a basic block. Therefore, the value can be discarded at the end of the basic block irrespective of whether it has been unspilled or not, and the spill area associated with it can be freed.

As we said above, the space for spilling registers is allocated in the local variable space in the stack for the subroutine being processed. The machine calling conventions (see calling conventions in [ConAs]) allocate local memory to a subroutine by subtracting the SP register by the amount of local space needed by the subroutine. This operation is done as the first instruction inside every subroutine. Therefore, the register manager has to compute the total "spill area" required by the subroutine. This is done by computing the "spill area" required for each basic block for which code is being generated and taking the maximum value for this over all the basic blocks in a subroutine. This "spill area" space is added to the local variable space already allocated by the translator to get the total local variable space required by the subroutine.

Since the local memory required for a subroutine is not known until the entire subroutine has been processed, the subtract operation at the beginning of the subroutine is emitted with a *symbolic name constant*, (see the assembler manual [ConAs]). The value of the symbolic name constant is defined using a pseudo-op after the entire subroutine has been processed.

3.6.4 Specific register assignment

There are three situations in which the register manager has to assign a specific register of a register class. They are:

1. Register S0 is required for a return value.
2. Register A5 is required by the **ldvi** and **stvi** instructions.
3. All vector reduction instructions operate on register pairs consisting of a vector register and a scalar register of the same number.

In cases 1 and 2 the specific register is made free (if it is not already so) by emitting suitable move instructions to copy the value onto another free register of the appropriate class. Then the specific register is allocated. In case 3 any available vector register is assigned first to the

register pair and then the corresponding scalar register is assigned in a manner identical to cases 1 and 2. This is done because assigning a specific vector register is a costly operation especially if it involves spilling a vector.

3.7 Transformer

As we said in section 3.1, the transformer shapes and canonicalizes the IR input to the pattern matcher. The shaping or restructuring is done to minimise the disadvantage of having a strict left-to-right evaluation order enforced by the pattern matcher. The canonicalizations are done in order to reduce the number of grammar productions needed, especially those describing the addressing modes. Also, this phase is crucial to ensure good quality of the code produced. Though this phase has not been implemented, the pattern matcher makes the following assumptions about this phase:

1. The IR does not allow root-level operators as internal nodes. The transformer should replace subtrees having root-level operators by compiler generated temporaries. The sub-tree is computed and stored first in the temporary.

2. The transformer should replace the operators in the translator output that are not provided in the IR. For example, the IR does not provide the full set of relational operators. The operators provided are less than, less than or equal to and equal to. Therefore other relational operators such as greater than and greater than or equal to have to be rewritten in terms of these operators. Also, the IR does not allow boolean assignments. Hence boolean assignments have to be rewritten as a sequence of trees of tests, jumps and ordinary assignments.

3. The transformer should do the following canonicalizations of the IR input. Firstly, subtraction by a constant, if possible, may be replaced by addition with the negative of the constant. This is to make possible the use of more complicated addressing modes. Secondly, constant operator of a commutative operation should be made the left child of the operator. Similarly, commutative operators with address constants (i.e., those address symbols specified with the **addr** symbol in the IR) should be made the left child except when the left child is another constant. These *canonicalizations* reduce the number of patterns in the target machine description, especially those

describing the addressing modes. The code generator can function without these canonicalizations, but may produce inferior code.

4. The transformer may apply algebraic transformations to shape the tree so that more complex addressing modes are selected by the pattern matcher. Though this is optional, it is crucial for good code quality.

5. Another optional transformation is to identify those subtrees that will always require register spilling on the machine. Such subtrees can be factored out of expressions and references to them replaced with compiler generated temporaries. This provides more registers for the computation of the subtree. But since the register manager can handle register spills on-the-fly this transformation too is optional.

Chapter 4

Conclusions

In this chapter we deal with the testing of the code generator, integration of the code generator with the rest of the compiler and compare the code generated with that of the existing CONVEX C220 FORTRAN compiler. We end the chapter by concluding the work.

4.1 Testing

Testing of the code generator mainly involved the following aspects.

1. The target machine description grammar.
2. The instruction selection phase that follows the pattern matcher.
3. The register allocation phase.
4. Overall testing of the code generator

Since the pattern matcher is automatically generated from the machine description grammar using a standard LALR(1) parser generator algorithm, correctness of the pattern matcher follows from that of the target machine description grammar. The grammar was checked to ensure that it was correct and that proper conflict resolution was being done. The report produced by the parser generator was useful in identifying productions in the grammar that were never reduced. This helped in fixing wrong conflict resolutions. The grammar was manually checked to be free of syntactic blocks. Since the machine does not provide many special case instructions, this checking was not difficult. Also, it was checked to see that productions were present for data transfer between the various registers and memory.

The checking of the instruction selection phase mainly consisted of checking that the instruction table had an entry for every instruction pattern selected by the pattern matcher. Since the table is consulted before each instruction is selected, its correctness ensured that illegal instructions are never generated by the code generator. Similarly, for each type of an operation that was valid in the IR, instruction table either had a corresponding instruction in the target machine, or had a function which checked the semantic restrictions and generated the default instruction sequences.

The testing of the register allocator and the overall testing of the code generator was done by running the code generator with various test inputs. The input consisted of the IR, symbol table information for all names and constants used in the IR, and the optional use count information for names and constants. These are supplied in three files as described in section 3.3.3. The register allocator was tested to check whether use counts were being correctly managed, register spilling and unspilling was correct and the various data structures were maintained consistently. The interface provided by the code generator was useful in the run-time checking of the address and register descriptors.

4.2 Integration

The code generator is assumed to produce code one basic block at a time. As was said in section 3.3.3 in the previous chapter, the code generator takes three input files. The first file gives the symbol table information from which the code generator builds a temporary symbol table. The file is hand generated. However, this symbol table should be created by the front-end of the compiler. Therefore, when the code generator is integrated with the rest of compiler, the symbol table created by the front end of the compiler can be used directly by the code generator. Hence, when this integration is done, there is no more need for creating the symbol table from the first input file. Hence, the routines in the code generator which read this file and build the temporary symbol table can and should be deleted. Routines which access the temporary symbol table should be changed to access the symbol table created by the front-end. These changes are confined to a few routines.

The second file contains the IR input in text format. Currently, this file is read by a lexical interface created using *flex*. Flex creates a function (yylex), which reads the input file and returns the internal code for the next symbol or token in the IR. Whenever the code

generator requires the next symbol, it invokes the function yylex. In the final integration, the IR is produced by the translator phase. Hence, yylex should be changed to read the IR symbols from memory locations where the translator would have previously written them. As in the case of the symbol table, now the IR input file and the lexical interface is no longer required.

Similarly, the third input file which gives the use count information of variables, can be removed in the final integration. An internal *use count table* is initialised by the code generator with the information in the file. This table can be directly initialised by the front-end.

4.3 Comparisons

The code generated by our code generator was compared with that produced by the existing FORTRAN compiler in the CONVEX machine (with optimization switches turned off for comparisons 1, 2, and 3). The following subsections give a few comparisons.

4.3.1 Comparison 1

Code is generated for the FORTRAN statement x(i) = 10 + y(i) + i. The assumptions made about the variables are

1. Arrays x and y and the variable i are of data type integer. The machine data type corresponding to integers is word (4 bytes).

2. Array indexing is from 1 onwards.

3. x is in a common block *_blnk_*. Variables y and i are local to the function f2, in which the expression is evaluated, and have been allocated memory on the run-time stack dynamically. Hence, y and i are addressed using the frame pointer fp.

The actual expression computed is:

$$-4 + X + 4* \uparrow (I + FP) = 10 + \uparrow (-4 + (Y + FP) + 4* \uparrow (I + FP)) + \uparrow (I + FP) \quad (4.1)$$

where X, Y and I are address constants. X stands for the address constant expression "*_blnk_ + offset of x within the common block*" and Y and I are offsets from the frame pointer. ' $\uparrow$' (ind) is the dereferencing operator. Note that the computation has

been rearranged keeping in view of the addressing modes allowed in the machine. This rearrangement has been done on the lines discussed in section 3.7. The constant -4 in the expression is to account for the starting of indexing from 1 onwards. The computation of the address of y, Y + FP, has been grouped together. This has been done to make better use of the addressing modes. Except for that, the add operation is treated as left associative. The IR of the code fragment is:

```
label: f2
st.w
     add.w add.w const.w: 10
           ind.w add.w add.w const.w: -4 add.w addr:y fp
           mul.w const.w: 4 ind.w add.w addr: i fp
     ind.w add.w addr: i fp
add.w add.w const.w: -4 addr: x mul.w const.w: 4 ind.w add.w addr: i fp
ret
```

The code produced by our code generator is shown in figure 4.1(a) and that of the CONVEX compiler in figure 4.1(b). The order of evaluation of the expression in the CONVEX compiler is different from the one used by our code generator.

The CONVEX FORTRAN compiler has generated 46 bytes of code with 14 instructions and using 3 address registers. Our code generator has generated 52 bytes of code comprising of 15 instructions and using 1 address and 2 scalar registers. The extra instruction generated is a **mov** instruction (line 14 in figure 4.1(a)) to move the contents of a scalar register to an address register. This is expected because our code generator always loads variables into scalar registers irrespective of whether the value occurs in an addressing context. Also, since no use count has been specified for the variable i, it is loaded afresh from memory for each of its use. Hence our code generator has three load instructions for i, (lines 5, 10 and 12 in figure 4.1(a)) while in figure 4.1(b), there is a single load and two moves instead (lines 3, 5 and 9 in figure 4.1(b)). This accounts for the extra bytes generated. If we specify a use count of three for i in our code generator, this inefficiency is removed. The code generated in this case is shown in figure 4.2.

Comparing figure 4.2 with figure 4.1(b) we find that both contain 14 identical instructions except for the difference in the class of the registers and the address constants. Our

```
1)  f2:
2)  sub.w   #f2_stack_space, sp
3)  ldea    -40(fp), a1          ; y
4)  add.w   #-4, a1
5)  ld.w    -44(fp), s0          ; i
6)  mul.w   #4, s0
7)  add.w   s0, a1
8)  ld.w    (a1), s0
9)  add.w   #10, s0
10) ld.w    -44(fp), s1          ; i
11) add.w   s0, s1
12) ld.w    -44(fp), s0          ; i
13) mul.w   #4, s0
14) mov     s0, a1
15) st.w    s1, _blnk_ + 0 + -4(a1)
16) rtn
17) f2_stack_space = 44
```

(a)

```
1)  _f2_:
2)  sub.w   #48,sp               ;#13
3)  ld.w    -4(fp),a5            ;#19, I
4)  ldea    -44(fp),a1           ;#19, Y
5)  mov     a5,a2                ;#19
6)  mul.w   #4,a2                ;#19
7)  add.w   #-4,a2               ;#19
8)  add.w   a2,a1                ;#19
9)  mov     a5,a2                ;#19
10) mul.w   #4,a2                ;#19
11) ld.w    0(a1),a1             ;#19, Y
12) add.w   a5,a1                ;#19
13) add.w   #10,a1               ;#19
14) st.w    a1,__blnk_-4(a2)     ;#19, X
15) rtn                          ;#21
```

(b)

Figure 4.1: Codes generated for the statement x(i) = 10 + y(i) + i

```
1)  f2:
2)  sub.w   #f2_stack_space, sp
3)  ldea    -40(fp), a1          ; y
4)  add.w   #-4, a1
5)  ld.w    -44(fp), s0          ; i
6)  mov.l   s0, s1
7)  mul.w   #4, s1
8)  add.w   s1, a1
9)  ld.w    (a1), s1
10) add.w   #10, s1
11) add.w   s0, s1
12) mul.w   #4, s0
13) mov     s0, a1
14) st.w    s1, _blnk_ + 0 + -4(a1)
15) rtn
16) f2_stack_space = 44
```

Figure 4.2: Code produced by our code generator for the statement x(i) = 10 + y(i) + i with a use count of 3 for i.

code generator produces a single load for i and only a single move (line 6 in figure 4.2) to preserve the value of i instead of the 2 moves (lines 5 and 9) in figure 4.1(b). For the second use of i in line 11 in figure 4.2, our code generator makes s0 (register s0 contains value of i) the first operand of the add operation. Hence the value of i is not destroyed by this use and a move operation is avoided. It should be noted that this is possible because the other operand of the add operation is also a scalar register.

4.3.2 Comparison 2

This example illustrates register spilling. The expression computed is

$$x = a/b + c/d + e/f + g/h + i/j + k/l + m/n + o/p + q \tag{4.2}$$

The expression is evaluated assuming '+' to be right associative. The IR input to the code generator is:

```
label: f3
st.w
add.w div.w ind.w addr.w: b ind.w addr.w: a
add.w div.w ind.w addr.w: d ind.w addr.w: c
add.w div.w ind.w addr.w: f ind.w addr.w: e
add.w div.w ind.w addr.w: h ind.w addr.w: g
add.w div.w ind.w addr.w: j ind.w addr.w: i
add.w div.w ind.w addr.w: l ind.w addr.w: k
add.w div.w ind.w addr.w: n ind.w addr.w: m
add.w div.w ind.w addr.w: p ind.w addr.w: o
ind.w addr.w: q
addr: x
ret
```

The code generated by our code generator is shown in figure 4.3 and that of the CONVEX compiler in figure 4.4. Figure 4.3 generates a spill in line 25. The spilled value is reloaded in line 36. Hence the code generated by our code generator has one extra store and load. The CONVEX compiler avoids spill by rearranging the computation. Similar rearrangement (i.e, a left recursive input) would prevent spills in our case too. But we have

```
1)  f3:
2)  sub.w   #f3_stack_space, sp
3)  ld.w    b, s0           ; b
4)  ld.w    a, s1           ; a
5)  div.w   s0, s1
6)  ld.w    d, s0           ; d
7)  ld.w    c, s2           ; c
8)  div.w   s0, s2
9)  ld.w    f, s0           ; f
10) ld.w    e, s3           ; e
11) div.w   s0, s3
12) ld.w    h, s0           ; h
13) ld.w    g, s4           ; g
14) div.w   s0, s4
15) ld.w    j, s0           ; j
16) ld.w    i, s5           ; i
17) div.w   s0, s5
18) ld.w    l, s0           ; l
19) ld.w    k, s6           ; k
20) div.w   s0, s6
21) ld.w    n, s0           ; n
22) ld.w    m, s7           ; m
23) div.w   s0, s7
24) ld.w    p, s0           ; p
25) st.l    s1, -8(fp)
26) ld.w    o, s1           ; o
27) div.w   s0, s1
28) ld.w    q, s0           ; q
29) add.w   s1, s0
30) add.w   s7, s0
31) add.w   s6, s0
32) add.w   s5, s0
33) add.w   s4, s0
34) add.w   s3, s0
35) add.w   s2, s0
36) ld.l    -8(fp), s2
37) add.w   s2, s0
38) st.w    s0, x           ; x
39) rtn
40) f3_stack_space = 8
```

Figure 4.3: Code produced by our code generator for the statement x = a/b + c/d + e/f + g/h + i/j + k/l + m/n + o/p + q

```
1)  _MAIN_:
2)  sub.w   #0,sp       ;#7
3)  ld.w    LU+60,s0    ;#7, O
4)  ld.w    LU+64,s1    ;#7, P
5)  ld.w    LU+52,s2    ;#7, M
6)  ld.w    LU+56,s3    ;#7, N
7)  div.w   s1,s0       ;#7
8)  ld.w    LU+44,s1    ;#7, K
9)  ld.w    LU+48,s4    ;#7, L
10) div.w   s3,s2       ;#7
11) ld.w    LU+36,s3    ;#7, I
12) ld.w    LU+40,s5    ;#7, J
13) div.w   s4,s1       ;#7
14) ld.w    LU+28,s4    ;#7, G
15) ld.w    LU+32,s6    ;#7, H
16) div.w   s5,s3       ;#7
17) ld.w    LU+20,s5    ;#7, E
18) div.w   s6,s4       ;#7
19) ld.w    LU+24,s6    ;#7, F
20) ld.w    LU+68,s7    ;#7, Q
21) div.w   s6,s5       ;#7
22) ld.w    LU+12,s6    ;#7, C
23) add.w   s7,s0       ;#7
24) ld.w    LU+16,s7    ;#7, D
25) div.w   s7,s6       ;#7
26) ld.w    LU+4,s7     ;#7, A
27) add.w   s0,s2       ;#7
28) ld.w    LU+8,s0     ;#7, B
29) add.w   s2,s1       ;#7
30) div.w   s0,s7       ;#7
31) add.w   s1,s3       ;#7
32) add.w   s3,s4       ;#7
33) add.w   s4,s5       ;#7
34) add.w   s5,s6       ;#7
35) add.w   s6,s7       ;#7
36) st.w    s7,LU       ;#7, X
37) rtn                 ;#9
```

Figure 4.4: Code produced by CONVEX FORTRAN compiler for the statement x = a/b + c/d + e/f + g/h + i/j + k/l + m/n + o/p + q

not taken into account the instruction pipelining provided by convex. The Convex compiler does this by two ways. First, it groups together load instructions so that a register value is not used immediately after loading (there are some other instructions between loading and use). Secondly, if a register is modified by one instruction, then it may not be modified by the next instruction also. We have completely ignored pipelining issues as the evaluation order is fixed in the code generator.

4.3.3 Comparison 3

Code generated for the statement k = x(i, j) is compared here. x is a 10 X 10 array of integers stored in column major order. x is passed by reference to the function which computes the above statement. i, j and k are integer local variables. Size of an integer is 4 bytes. All indexes of the array x begin from 1 onwards. Based on the above assumptions, the actual expression to be computed is derived using a standard array translation method (see [ASU86]). The expression computed is:

$$K + FP = \uparrow(-44 + \uparrow(X + AP) + 40 * \uparrow(J + FP) + 4 * \uparrow(I + FP)) \qquad (4.3)$$

where K, X, I and J are address constants corresponding to variables k, x, i and j. In the IR, the transformations are applied. Addition is assumed to be left associative. The IR is:

```
label: f12
st.w ind.w add.w add.w add.w const.w: -44
                          ind.w add.w addr: x ap
                    mul.w const.w: 40 ind.w add.w addr: j fp
              mul.w const.w: 4 ind.w add.w addr: i fp
add.w addr: k fp
ret
```

The code generated are shown in figures 4.5(a) and 4.5(b). The number of instructions are 13 and 11 for our code and CONVEX compiler code respectively. Register usage is 2 scalar and 1 address registers for our code generator while it is 3 address and 1 scalar for the CONVEX compiler. Thus, the number registers used in our code is one less than the CONVEX compiler code. The code generated by our code generator has two extra instructions, those of lines 4 and 7 in figure 4.5(a). The extra move instruction on line 7 is

```
1)  f12:
2)  sub.w   #f12_stack_space, sp
3)  ldea    -44(ap), s0         ; x
4)  add.w   #-44, s0
5)  ld.w    -8(fp), s1          ; j
6)  mul.w   #40, s1
7)  add.w   s0, s1
8)  ld.w    -4(fp), s0          ; i
9)  mul.w   #4, s0
10) add.w   s1, s0
11) mov     s0, a1
12) ld.w    (a1), s0
13) st.w    s0, -12(fp)         ; k
14) rtn
15) f12_stack_space = 12
```

(a)

```
1)  _f2_:
2)  sub.w   #16,sp              ;#12
3)  ld.w    -4(fp),a5           ;#17, I
4)  ld.w    -8(fp),a1           ;#17, J
5)  mul.w   #4,a5               ;#17
6)  ld.w    0(ap),a2            ;#17, X
7)  mul.w   #40,a1              ;#17
8)  add.w   a1,a5               ;#17
9)  add.w   a5,a2               ;#17
10) ld.w    -44(a2),s0          ;#17, X
11) st.w    s0,-12(fp)          ;#17, K
12) rtn                         ;#18
```

(b)

Figure 4.5: Codes produced for the statement k = x(i, j)

again because of using scalar registers to compute in an addressing context. The other extra instruction is an explicit addition of the constant -44 in line 7. The CONVEX compiler subsumes this add in the address computation on line 10 in figure 4.5(b). Our code generator fails to detect this because of the decision to treat the addition operation as left associative. The two extra instructions result in 6 bytes of extra storage. If the transformer is able to shape the input tree in the following way, then this add operation can be subsumed.

$$K + FP = \uparrow(-44 + (\uparrow(X + AP) + 40 * \uparrow(J + FP) + 4 * \uparrow(I + FP))) \qquad (4.4)$$

Here the add operation is assumed to be right associative. The parentheses indicate the order of evaluation , i.e., the shape of the IR tree. An additional difference in the two code sequences is that our code generator has used a ldea instruction with an indirection while the CONVEX compiler used the equivalent ld.w [1](see lines 3 and 6 respectively in the two code sequences).

4.3.4 Comparison 4

In this comparison, we deal with the code generated for two vector statements. First is x = a + b * c where x, a, b and c are all vectors of length 100. x, a, b, c are of integer type and are allocated space in the common area 'comm1'. The second is a vector reduction statement j = sum(x) where j is an integer. Sum computes the sum of the elements of the vector x and *adds* this sum to j. Hence the need to initialise the contents of j. Convex provides a single instruction to find this sum which computes the sum of a vector register Vn and adds this to the scalar register Sn where n is a register number. This example was run with optimization flags switched on for the CONVEX FORTRAN compiler in order to force it to apply vectorization. The IR is:

```
label: f19
mov.w const.w: 100 vl
mov.w const.w: 4 vs
st.w add.w vecind.w addr: a
     mul.w vecind.w addr: b vecind.w addr: c
addr:x
```


[1]ldea is more intuitive since the value loaded is the address of x.

```
1)  f19:
2)  sub.w   #f19_stack_space, sp
3)  ld.w    #100, vl
4)  ld.w    #4, vs
5)  ld.w    comm1 + 800, v0    ;b
6)  ld.w    comm1 + 1200, v1   ;c
7)  mul.w   v0, v1, v1
8)  ld.w    comm1 + 400, v0    ;a
9)  add.w   v0, v1, v1
10) st.w    v1, comm1 + 0      ;x
11) ld.w    #1, s0
12) st.w    s0, comm1 + 1600   ;j
13) ld.w    comm1 + 0, v1      ;x
14) ld.w    comm1 + 1600, s0   ;j
15) mov.l   s0, s1
16) sum.w   v1
17) st.w    s1, comm1 + 1600   ;j
18) rtn
19) f19_stack_space = 0
```

(a)

```
1)  _f19_:
2)  ld.w    #1,s0                ;#17
3)  st.w    s0,__comm1_+1600     ;#17, J
4)  ld.w    #100,vl              ;#21
5)  ld.w    __comm1_+1600,s0     ;#20, J
6)  ld.w    #4,vs                ;#19
7)  ld.w    __comm1_+800,v0      ;#19, B
8)  ld.w    __comm1_+1200,v1     ;#19, C
9)  ld.w    __comm1_+400,v2      ;#19, A
10) mul.w   v0,v1,v3             ;#19
11) add.w   v3,v2,v0             ;#19
12) st.w    v0,__comm1_          ;#19, X
13) sum.w   v0                   ;#20
14) st.w    s0,__comm1_+1600     ;#20, J
15) rtn                          ;#23
```

(b)

Figure 4.6: Codes produced for vector operations x = a + b * c and j = sum(x)

```
st.w const.w: 1 addr: j
st.w sum.w vecind.w addr: x const.w: 0 addr: j
ret
```

The code generated are shown in figures 4.6(a) and 4.6(b). The code generated by our code generator has three extra instructions, those of lines 13, 14, and 15 in figure 4.6(a). Lines 13 and 14 are because the values of x and j in registers are not being reused. The CONVEX compiler detects these reuses and avoids reloads. This can be effected in our code generator by specifying a use count of 1 each for both j and x. The extra instruction on line 15 is a move. This is because x and j have to be in same numbered registers (v1 and s1) for the sum operation as said in the beginning of this subsection. The mul instruction on line

10 in the code generated by the CONVEX compiler (figure 4.6(b)) has three distinct vector registers for source and destination (v0 and v1 are sources and v2 is the destination). This is to utilise the possible pipelining of vector instructions. There are two constraints imposed for pipelined execution. The first is that the vector registers have been grouped into four groups (v0, v4), (v1, v5), (v2, v6) and (v3, v7). Each group allows only 2 reads and 1 write at a time. If these limits are exceeded for a group of instructions then these instructions are executed sequentially. Our on-the-fly register allocation strategy has not taken this into consideration. However, it should be noted that the CONVEX compiler generated code has used 2 more vector registers compared to our code generator. Secondly, there are only three functional units to implement all the vector operations. Hence if two instructions require the same functional unit, (for example **add** and **sub** instructions), the instruction execution is again sequential. Hence the adjacency of instructions which require different functional units should be encouraged for pipelining. This reordering is not possible within the code generator since the evaluation order is fixed.

4.3.5 Comparison 5

Here we consider the vector code generated for the FORTRAN do-loop shown below. x, y, a, b, c, are integer vectors of length 100. y is a local variable while all the others are in common block comm1. The CONVEX FORTRAN compiler was run with the optimisation flags switched on. This comparison illustrates the use of extended vector instructions, i.e., those using the 't' or 'f' masks. It also illustrates the transformation of the $'>'$ relational operator (which is not available in the machine) to the $'<'$ operator with the position of operands swapped.

```
do 10 I = 1, 100, 1
if (x(i) .GT. 0) then
y(i) = a(i) + b(i) + c(i) / x(i)
endif
10 continue
```

The IR input to our code generator is:

```
label: f21
mov.w const.w: 100 vl
mov.w const.w: 4 vs
veclt.w const.w: 0 vecind.w addr: x
st.w.t
        vecadd.w.t vecadd.w.t vecind.w.t addr: a vecind.w.t addr: b
                   vecdiv.w.t vecind.w.t addr: c vecind.w.t addr: x
add.w addr: y fp
ret
```

The code generated by our code generator and CONVEX are shown in figure 4.7(a) and 4.7(b) respectively. Our code generator output has 15 instructions to the 14 of the CONVEX FORTRAN compiler. The extra instruction is the reloading of x (line 12 in figure 4.7(a)). The CONVEX compiler reuses the value of x already in a register. Specifying a use count of 2 for x in our code generator removes this extra instruction. Again, our code generator has ignored pipelining issues, as in the previous example. Hence, the number of vector registers used is 3, one less than that used by the CONVEX compiler.

4.4 Summary of comparisons

Figure 4.4 summarises the comparisons discussed in the preceding section. The number of instructions (the *instructions* field in the table), number of bytes generated (the *Bytes* field), and the number of registers used for each register class (fields *Areg, Sreg and Vreg*, respectively for address, scalar and vector registers) are shown. A comparison with an asterisk indicates that the comparison was done with use counts specified to our code generator.

```
1)  f21:
2)  sub.w   #f21_stack_space, sp
3)  ld.w    #100, v1
4)  ld.w    #4, vs
5)  ld.w    #0, s0
6)  ld.w    comm1 + 0, v0       ;x
7)  lt.w    s0, v0
8)  ld.w    comm1 + 400, v0     ;a
9)  ld.w    comm1 + 800, v1     ;b
10) add.w.t v0, v1, v1
11) ld.w    comm1 + 1200, v0    ;c
12) ld.w    comm1 + 0, v2       ;x
13) div.w.t v0, v2, v2
14) add.w.t v1, v2, v2
15) st.w.t  v2, -400(fp)        ;y
16) rtn
17) f21_stack_space = 400
```

(a)

```
1)  _sub1_:
2)  sub.w   #424,sp             ;#13
3)  ld.w    #100,v1             ;#23
4)  ld.w    #0,s0               ;#21
5)  ld.w    #4,vs               ;#20
6)  ld.w    __comm1_,v0         ;#20, X
7)  lt.w    s0,v0               ;#20
8)  ld.w    __comm1_+1200,v1    ;#21, C
9)  ld.w    __comm1_+400,v2     ;#21, A
10) div.w.t v1,v0,v3            ;#21
11) add.w.t v3,v2,v0            ;#21
12) ld.w    __comm1_+800,v1     ;#21, B
13) add.w.t v1,v0,v2            ;#21
14) st.w.t  v2,-400(fp)         ;#21, Y
15) rtn                         ;#25
```

(b)

Figure 4.7: Comparison 5: Code produced for a FORTRAN do-loop

Comparison	Instructions		Bytes		Areg		Sreg		Vreg	
	our	convex	our	convex	our	convex	our	convex	our	convex
1	15	14	52	46	1	3	2	0	0	0
1*	14	14	46	46	1	3	2	0	0	0
2	38	36	118	110	0	0	8	8	0	0
3	13	11	44	38	1	3	2	1	0	0
4	17	14	58	48	0	0	2	1	2	4
4*	15	14	50	48	0	0	2	1	2	4
5	15	14	66	62	0	0	1	1	3	4
5*	14	14	62	62	0	0	1	1	3	4

Figure 4.8: Comparison Summary

4.5 Conclusions

The code generation phase of the the ongoing FORTRAN-90 compiler project has been developed using an automatic table driven code generation method. By virtue of the method used, the code generator will be easy to maintain and tune when it is integrated with the rest of the compiler. As the comparisons show, the code generated is comparable in quality with the existing FORTRAN compiler on the CONVEX C220 machine. However, pipelining facilities provided by the machine for vector operations could not be exploited while generating code. This issue of pipelining in the context of automatic code generation methods needs to be explored.

Bibliography

[AGH84] Aigrain, P, Graham, S. L., Henry, R. R., McKusick, M. K. and Pelegri-Llopart, E., *"Experience with a Graham-Glanville style code generator"*, Proceedings of ACM SIGPLAN 1984 symposium on Compiler Construction, SIGPLAN Notices 19, 6 (Jan. 1984) 13-24.

[ALK87] Allen, R., and Kennedy K., *"Automatic translation of FORTRAN programs to vector form"*, ACM TOPLAS Vol. 9, No. 4, Oct. 1987.

[ASU86] Aho, A. V., Sethi, R., and Ullman, J. D., *"Compilers : Principles, Techniques and Tools"*, Addison Wesley, Reading , MA, 1986.

[Catt80] Cattell, R. G. G., *"Automatic derivation of code generators from machine descriptions"*, ACM TOPLAS 2, 2 (Apr, 1980), 173-190.

[ConArc] *"CONVEX Architecture reference"*, CONVEX Computer Corporation, 1988.

[ConAs] *"CONVEX Assembly language user's guide"*, CONVEX Computer Corporation, 1988.

[GFH82] Ganapathi, M., Fischer, C. N. and Hennessy, T. L. *"Retargetable Compiler code generation"*, ACM Computing Surveys 14, 4 (Sep. 1982), 573-592

[GHS82] Graham, S. L., Henry, R. R., Schulman R. A., *"An experiment in table driven code generation"*, Proceedings of the ACM SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices, 17, 6 (Jun. 1982), 32-43.

[GlG78] Glanville, R. S., and Graham S. L., *"A new method for compiler code generation"*, Proceedings, 5th ACM symposium on Principles of Programming Languages, Tuscon, AZ, Jan. 1978, 231-240.

[Glan77] Glanville, R. S., *"A machine independent algorithm for code generation and its use in retargetable compilers"*, Ph.D dissertation, Departments of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Dec 1977.

[Grah80] Graham, S. L., *"Table-driven code generation"*, IEEE Computer 13, 8 (Aug. 1980), 25-34.

[HB85] Kai Hwang, Faye A. Briggs *"Computer Architecture and Parallel Processing"*, McGraw Hill, Singapore 1985.

[Hen86] Henry, R. R., *"Code generation tutorial"* , ACM SIGPALN Compiler Construction Conference Jun. 1986, Palo Alto, CA.

[HoS90] Horowitz, E., Sahini, S., *"Fundamentals of Computer Algorithms"*, Galgotia Publication, 1990.

[HoU87] Hopcroft, E. J., Ullman, J. D., *"Introduction to Automata theory Languages and Computation"*, Addison-Wesley , 1987.

[John75] Johnson, S. C., *"YACC: Yet another compiler compiler"*, Computing Science Technical Report 32, AT & T Bell Laboratories, 1975.

[John77] Johnson, S. C., *"A tour through the portable C compiler"*, UNIX documentation, Bell Telephone Laboratories, Murray Hill, N. J., 1977.

[John78] Johnson, S. C., *"A portable compiler: Theory and Practice"*, Proceedings of the ACM Symposium on Principles of Programming Languages, Tuscon, AZ. Jan. 1978, 97-104.

[KM86] Kumar, S., and Malhotra, V. M., *"Retargetable Compiler code generation: A new technique"*, FST & TCS6 Conference, LNCS Vol. 241 Springer Verlag, 1986, 57-80.

[Les75] Lesk, M. E., *"Lex: A lexical analyzer generator"*, Computing Science Technical Report, No. 39, Bell Labs, Murray Hill, NJ Oct. 1975.

[NSP86] Nori, K.V., Sanjeev Kumar, Pawan Kumar, M., *"Retrospection on the PQCC compiler structure"*, FST & TCS Conference, LNCS Vol. 237 Springer Verlag, 1988, 501-527.

[Wulf75] Wulf, W., Johnson, R. K., Weinstock, C. B., Hobbs, S. O., and Geschke, C. M., *"The Design of an Optimizing Compiler"*, American Elsevier Computer Science Library, 1975.

[Wulf80] Wulf, W., Leverett, B. W., Cattell, R. G. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., and Schatz B. R., *" An overview of the production quality compiler-compiler project"*, IEEE Computer, 13, 8 (Aug, 1980), 38-49.

Appendix A

Target Machine Description Grammar

This appendix gives the target machine grammar describing the CONVEX-C220 machine. The grammar is in the form input to the parser generator. Please refer to [John75] for the parser (yacc) input syntax.

```
%start s /* The start symbol of the grammar */
/*
    The operators in the IR.  Please refer to the CONVEX
    architecture and assembly manuals [ConArc] and [ConAs]
    for the details regarding these operators.  The operator IND
    below is not explicitly present in the instruction set.
    IND operator implicitly stands for the load instruction
    of the machine.
*/
%token <OP_U>   ADD ALL AND ANY ATAN CALL CALLQ CALLS COS CPRS CVTB CVTD
                CVTH CVTL CVTS CVTW DIV EQ EXP FRINT IND JBR LDVI LE LEU
                LN LOP LT LTU MASK MAX MERG MIN MOV MUL NEG NOT OR POP
                PROD PSH PSHEA RET RETQ SHF SIN SQRT ST STE STVI SUB SUM
                VECADD VECAND VECCVTB VECCVTD VECCVTH VECCVTL VECCVTS
                VECCVTW VECDIV VECEQ VECFRINT VECIND VECLE VECLOP VECLT
                VECMUL VECNOT VECOR VECSHF VECSQRT VECSUB VECXOR XOR XPND
```

```
%token <OP_U> RETVAL /* Represents the value returned
                        by a function call
                     */
/* The input operand terminals */
%token <ID_U> ADDR /* address constant */
%token <CONST_U> CONST /* ordinary constant valid in the assembler */
%token <ID_U> LABELL /* a label */
%token <ID_U> VAR /* short form of IR expression "ind addr" */
%token <ID_U> VEC /* short form of IR expression "vecind addr" */
%token <reg_U> Sreg /* to specify a scalar register in the input */
%token <reg_U> Areg /* to specify an address register in the input */
%token <reg_U> Vreg /* to specify a vector register in the input */
%token <reg_U> AP FP SP VM VMU VML VL VS VLS /* The special registers */
/* Nonterminals representing the register classes.  */
%type <reg_U> areg sreg vreg
/* Non-terminals for grouping operators */
%type <OP_U> call_op ret_op op_set_2 op_set_3 op_set_4 op_set_5
             op_set_6 op_set_8 op_set_9 op_set_10 op_set_11
             op_set_12 op_set_13 op_set_14 op_set_16
%type <effa_U> effa /* non_terminal for factoring addressing modes. */
%type <reg_U> ded_addr_reg /* ded_addr_reg represents AP, SP or FP */
/* non-terminals for maintaining the address and register
   descriptors.
 */
%type <OP_U> ind /* Corresponds to terminal IND */
%type <ID_U> addr /* Corresponds to terminal ADDR */
%type <CONST_U> constant /* Corresponds to terminal CONST */
%type <ID_U> label /* Corresponds to terminal LABELL */

/* non-terminals introduced for parsing purposes */
%token <ival_U> YYD /* to turn on/off debugging */
%type <ival_U> label_tree
```

```
%type tree
%%
/*
   s generates a sequence of optionally labelled IR trees.
*/
s               : /* empty */
                | s label_tree
/*
   Each label_tree generates an optionally labelled tree with a
   root-level operator as the root of the tree.
*/
label_tree      : label tree
                | tree
                | YYD
                | error
                ;
label           : LABELL
                ;
/* tree derives all the root level operations. */
tree            : ST areg effa
                | ST sreg effa
                | ST vreg effa
                | ST VLS effa
                | ST VM effa
                | MOV sreg VMU /* values moved into special registers do
                                  not return any values. These operations
                                  simply set the special registers.
                               */
                | MOV sreg VML
                | MOV sreg VS
                | MOV sreg VL
                | MOV areg VS
```

```
                | MOV areg VL
                | MOV ind effa VM
                | MOV ind effa VLS
                | MOV constant VL
                | MOV constant VS
                | MOV ded_addr_reg ded_addr_reg
                | STE sreg effa
                | STVI sreg vreg
                | call_op effa
                | ret_op sreg
                | ret_op
                | JBR effa
                | PSH sreg
                | PSH areg
                | PSHEA effa
                | PSHEA constant
                | ADD constant ded_addr_reg
                | op_set_16 constant sreg
                | op_set_16 constant areg
                | op_set_16 sreg sreg
                | op_set_16 areg areg
                | op_set_12 vreg vreg
                | op_set_12 sreg vreg
                ;
/*
   sreg derives instructions which leave the value in a scalar
   register.  These productions are not together in order to have
   correct conflict resolution.
*/
sreg            : ind effa
                | Sreg
                ;
```

```
/*
  The following effa productions first move the sreg to an areg
  and then form an effa permitted by the machine.
 */
effa            : ind ADD ADD constant addr sreg
                | ADD ADD constant addr sreg
                | IND ADD constant ADD addr sreg
                | ADD constant ADD addr sreg
                | ind ADD constant sreg
                | ADD constant sreg
                | ind ADD addr sreg
                | ADD addr sreg
                ;
sreg            : ADD constant sreg
                | op_set_2 constant sreg
                | ADD sreg sreg
                | op_set_2 sreg sreg
                | op_set_3 sreg
                | op_set_4 sreg
                | op_set_5 sreg
                | op_set_6 sreg
                | op_set_8 sreg
                | op_set_9 vreg sreg
                ;
 sreg           : RETVAL
                | POP
                | VAR
                ;
 call_op        : CALL
                | CALLQ
                | CALLS
                ;
```

```
ret_op          : RET
                | RETQ
                ;
areg            : ADD areg areg
                | op_set_2 constant areg
                | op_set_2 areg areg
                | op_set_3 areg
                | ADD sreg areg
                | ADD areg sreg
                ;
vreg            : VEC
                | op_set_10 vreg sreg
                | op_set_11 vreg sreg
                | op_set_13 vreg sreg
                | op_set_10 vreg vreg
                | op_set_11 vreg vreg
                | op_set_13 vreg vreg
                | op_set_14 vreg vreg
                | op_set_11 sreg vreg
                | op_set_13 sreg vreg
                | VECIND effa
                | Vreg
                ;
 effa           : ind ADD ADD constant addr areg
                                /* effa = @ constant + addr(areg) */
                | ADD ADD constant addr areg
                                /* effa =  constant + addr(areg) */
                | ind ADD ADD constant addr ded_addr_reg
                            /* effa =  @ constant + addr(ded_addr_reg) */
                | ADD ADD constant addr ded_addr_reg
                            /* effa =  constant + addr(ded_addr_reg) */
```

```
| ind ADD constant ADD addr areg
        /* effa = @ constant + addr(areg) */
| ADD constant ADD addr areg
        /* effa =  constant + addr(areg) */
| ind ADD constant ADD addr ded_addr_areg
        /* effa =  @ constant + addr(ded_addr_reg) */
| ADD constant ADD addr ded_addr_areg
        /* effa =  constant + addr(ded_addr_reg) */
| ind ADD constant areg
        /* effa =  @ constant(areg) */
| ind ADD constant addr
        /* effa =  @ addr + constant */
| ADD constant addr
        /* effa =  addr + constant */
| ADD constant areg
        /* effa =  constant(areg) */
| ind addr
        /* effa =  @ addr */
| addr
        /* effa =  addr */
| ind ADD addr ded_addr_reg
        /* effa =  @ addr(ded_addr_reg) */
| ind ADD addr areg
        /* effa =  @ addr(areg) */
| ind areg
        /* effa = @ (areg) */
| ADD addr ded_addr_reg
        /* effa =  addr(ded_addr_reg) */
| ADD addr areg
        /* effa =  addr(areg) */
| areg   /* effa = (areg) */
;
```

```
areg            : ADD constant areg
                | ADD addr ded_addr_reg
                | ADD addr areg
                ;
/*
  Instructions to move a register into a scalar register.
 */
sreg            : areg
                | VMU
                | VML
                | VS
                | VL
                ;
/*
  Instructions to move a register into an address register which
  is not dedicated.
 */
areg            : sreg
                | addr
                | ded_addr_reg
                | Areg
                | VS
                | VL
                ;
 /*
   ded_addr_reg factors all the three special address registers
   FP, AP and SP.
  */
 ded_addr_reg    : FP
                 | AP
                 | SP
                 ;
```

```
ind             : IND
                ;
addr            : ADDR
                ;
constant        : CONST
                ;
op_set_2        : AND
                | DIV
                | MUL
                | OR
                | SHF
                | SUB
                | XOR
                ;
op_set_3        : NEG
                | NOT
                ;
op_set_4        : CVTW
                | CVTB
                | CVTH
                ;
op_set_5        : CVTS
                | CVTD
                | CVTL
                ;
op_set_6        : FRINT
                | LOP
                ;
op_set_8        : ATAN
                | COS
                | SIN
                | SQRT
```

```
                | EXP
                | LN
                ;
op_set_9        : MAX
                | MIN
                | PROD
                | SUM
                | ALL
                | ANY
                ;
op_set_10       : VECADD
                | VECAND
                | MASK
                | MERG
                | VECMUL
                | VECOR
                | VECXOR
                ;
op_set_11       : VECDIV
                | VECSUB
                ;
op_set_12       : VECLE
                | VECLT
                | VECEQ
                ;
op_set_13       : VECSHF
                ;
op_set_14       : VECCVTW
                | VECCVTS
                | VECCVTD
                | VECCVTB
                | VECCVTH
```

```
                | VECCVTL
                | CPRS
                | VECFRINT
                | LDVI
                | VECLOP
                | VECNOT
                | VECSQRT
                | XPND
                ;
op_set_16       : LE
                | LT
                | EQ
                | LEU
                | LTU
                ;
```